Chapter 15
Second-Generation Product Line Engineering: A Case Study at General Motors

Rick Flores, Charles Krueger, and Paul Clements

What you will learn in this chapter

- *An introduction to the basic concepts of the factory paradigm of product line engineering, including feature declarations, feature profiles, shared assets, variation points, and configurator*
- *The characterization of first-generation vs. second-generation product line engineering (2GPLE)*
- *How 2GPLE is being applied at General Motors and why the 2GPLE concepts have been critically important: How it has led to the creation of new roles and responsibilities, how organizational units at different levels and in different domain areas are cooperatively building PLE models that will all work together to define a vehicle*

1 Introduction

This chapter is the story of a product line engineering effort under way at General Motors. The product line involves the electronic control systems placed aboard vehicles during manufacturing. These control systems include electrical components (sensors and actuators), electronic control units laid out in a given topology around the car, wires and data networks to connect the components appropriately, and the software that runs it—all loaded correctly onto each vehicle. This story focuses on a particular set of aspects:

R. Flores
General Motors, Detroit, MI, USA
e-mail: rick.r.flores@gm.com

C. Krueger (✉) • P. Clements
BigLever Software, Austin, TX, USA
e-mail: ckrueger@biglever.com; pclements@biglever.com

- How solving this product line engineering problem requires every dimension of what has come to be called the *second-generation approach* to product line engineering.
- How a very small but consistent set of product line constructs are proving to be adequate to provide the necessary expressive power for this product line.
- How the automation that is required to power the product line solution depends not only on its own technical capabilities but also on vendor business partnerships that allow it to work seamlessly with a variety of lifecycle engineering tools that store artifacts in proprietary formats—artifacts that need to have variation points injected into them.

These aspects are made compelling because of the unprecedented complexity involved in this product line. If these solutions work here, it is unlikely they will be found wanting anywhere else.

2 Overview of Product Line Engineering

Systems and software product line engineering, often abbreviated as *product line engineering* (PLE), refers to the disciplined production of a portfolio of related products using a shared set of assets and a common means of production. The products in the portfolio are related by the features they have in common with each other; the variations among the products are also expressed as variations in the features they offer. The products can be

- Software
- Systems in which software runs or
- Non-software products that have software-representable artifacts (such as requirements, engineering models, or development plans) associated with them

In all cases, PLE works with the "soft" artifacts associated with the products and their production. PLE, then, includes and extends software product line engineering.

The key strategy behind PLE is to capitalize on commonality and manage variation in order to reduce the time, effort, cost, and complexity of creating and maintaining a product line of similar software systems:

- Capitalize on commonality through consolidation and sharing within the asset inputs, thereby avoiding duplication and divergence.
- Manage variation by clearly defining the variation points and decision model for exercising the variation points, thereby making the location, rationale, and dependencies for variation explicit.

The essence of PLE—for systems, software, and for manufacturing—is the focus on the single system rather than the many products. The "system" in this case consists of the *production line*, which enables the rapid production of any version of any of the products in the portfolio. A production line consists of a collection of software *assets*, a set of *feature profiles* that define the products, and the *configurator* that applies a feature profile to the assets in order to produce each product in the portfolio. Once the production line is established, products are instantiated rather than manually created.

PLE stands in contrast to classical product-centric development, in which each individual product is developed and evolved independently from other products, or (at best) starts out as a cloned copy of a similar product that is then changed to suit the new product's specific needs. Product-centric development takes very little advantage of the commonalities among products in a portfolio.

3 Basic PLE Concepts: The Factory Paradigm

PLE can be described in terms of the following five concepts:

- *Feature declarations* are parameters that express the diversity in a product line. Feature declarations are analogous to the choices that are available to you when you buy a new car: Two door or four door? Sport package, luxury package, or economy package? Moon roof? Feature declarations typically express the customer-visible diversity among the products in a product line.
- *Feature profiles* are used to select and assign values to the feature declaration parameters for the purpose of instantiating a product. A feature profile is associated with each product and reflects the actual choices you make: Two door with sport package but no moon roof or four door with luxury package and moon roof.
- *Systems and software assets* are configurable artifacts—such as models, source code, requirements, and test cases—engineered to be shared across the product line. They are the building blocks of the products in the product line. Assets can be whatever assets are representable with software and either compose a product or support the creation of a product.
- *Variation points* define the variations in the system and software assets used to build products. Feature declarations are mapped to these variation points, and a feature profile is mapped to the choices made at each variation point when building a product.
- *Configurator* is the automation that takes the feature choices reflected in a feature profile for a product and applies them to the variation points in the assets, so as to produce instances of the assets that are the building blocks of the product being built. It is possible to perform this step manually, but the task quickly becomes unmanageable without an automated tool. An example of an industry-leading configurator is Gears by BigLever Software [8], which GM chose to power its PLE effort.

An analogy with factory-base manufacturing serves to illuminate the concepts. Manufacturers have long used analogous engineering techniques to create a product line of similar products using a common factory that assembles and configures parts designed to be reused across the varying products in the product line. For example, automotive manufacturers can now create thousands of unique variations of one car

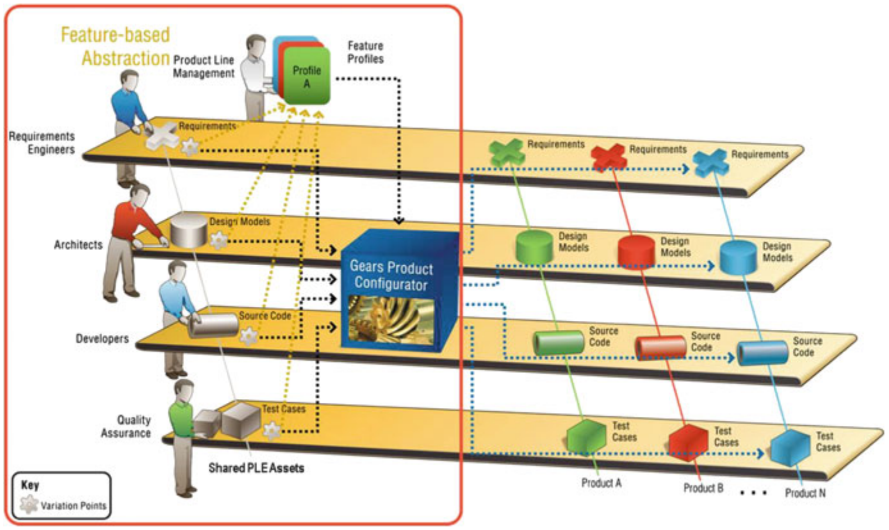


Fig. 15.1 A production line. Feature profiles drive instantiation of assets' variation points, which are exercised by the configurator (here, Gears) to produce product-ready instances

model using a single pool of carefully architected parts and one factory specifically designed to configure and assemble those parts.

In PLE, the configurator is the factory and the assets represent the factory's supply chain. Figure 15.1 illustrates.

4 First-Generation PLE

Product line engineering now has roots that span at least five decades, going back at least as far as Parnas's seminal paper on product families for software in 1976 [11]. Examining the rich historical legacy of this community reveals patterns of evolution in the state of the art and practice.

"Generations" are hard to pin down precisely and do not have rigid boundaries, but that does not prevent the concept from being useful. We can identify the Baby Boomer, Gen-X, Gen-Y, Tween, and Millennium Generations. Fighter aircraft are generally thought to be in their fifth generation [6] and programming languages in their fourth or fifth (opinions vary) [21]. Current standards for mobile broadband devices are known as 4G.

In the same spirit, we characterize some of the early and long-standing approaches to product line engineering as first generation. First-generation PLE (1GPLE) includes, among other things:

- A strong dichotomy between domain engineering and application engineering, or core asset development and product development. Application engineering

(or product development) is often said to include creating any assets used in a single product and promoting them to core assets only if subsequently used in more.
- Explicit inclusion of non-software artifacts in the collection of core assets, but with an unmistakable emphasis on software (under the umbrella of an all-encompassing software architecture) as the principal kind of core asset.
- Focus on features as the language to describe a product line's domain and a way to discriminate products from each other.
- Acknowledgment of configuration management as an essential practice under PLE but without a strong distinction between core asset CM and product CM.

These approaches have yielded a rich legacy of product line success, as evidenced by a plethora of case studies [3, 4, 9, 13, 16]. The newer approaches we describe in this chapter build on them. These newer approaches came about because of situations where more robust methods are needed to (among other things) deal with very large-scale product lines. "Scale" can refer to size, complexity, and number in terms of products, core assets, lifecycle phases involved, and evolutionary revisions over time.

5 Second-Generation PLE

PLE has been evolving a new set of concepts and technology that has been referred to as *second-generation product line engineering (2GPLE)*. This characterization represents seen-in-practice extensions to the earlier paradigm that was centered mainly on core asset production and product derivation.

Second-generation PLE can be said to comprise five aspects. None of these facets of 2GPLE are incompatible with or contradict earlier approaches to software product line engineering [4, 13, 19]—indeed, all five are mentioned as possible. The difference is that in 2GPLE they have emerged in a central role, essential to support large-scale practice. The five facets of 2GPLE are:

- Reliance on features as the *lingua franca* to express product differences in all phases of the life cycle
- Consistent and traceable variation management in artifacts across the full engineering life cycle
- A simplified configuration management model that maintains versioning of assets, not products or asset instantiations
- Feature models with encapsulating constructs to facilitate hierarchical product lines and cooperative feature model development across organizational boundaries
- Industrial-strength automation

GM could not accomplish its product line engineering goals without each one of these. We will discuss each in turn here and show how each is put into play at GM in the second half of this chapter.

5.1 *Features as the* Lingua Franca *to Express Product Differences Across the Life Cycle*

The concept of a feature as applied to families of software systems is thought to date to feature-oriented domain analysis methods, beginning with FODA [7]. In that work, the authors adopted the definition of feature right out of an ordinary dictionary: “A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems,” and this definition still serves us well in the 2GPLE world. The property of being visible to the user is perhaps the central notion; the choices a buyer can make when purchasing a new car is a helpful analogy.

Referring again to the manufacturing paradigm, the set of features to be exhibited by the product under construction drives the manufacturing process: The features determine which parts should be used in the product, how they should go together, and how they should be tailored to fit the product. All of the assets that go into building products will include variation points that will be exercised based on the features the product under construction needs to have.

The concept of “feature” allows a consistent abstraction to be employed when making choices from vehicle configuration all the way down to the deployment of software components onto an electronics architecture. As we will see, GM is elevating what they call a *bill-of-features* to the role of communication vehicle between business, product marketing, and engineering units. The goal is to use this to express and automatically derive content for vehicles in terms of desired features and capabilities, rather than describing vehicles in terms of its bill-of-materials—that is, its listing of parts and pieces. Although a bill-of-materials will still be needed for manufacturing, the vision of GM’s PLE effort is that the bill-of-materials for a vehicle’s electronics is generated from its bill-of-features.

The product line literature is rife with feature modeling languages and constructs, few of which have seen industrial application. The GM experience is providing a compelling argument that a very small and simple set of feature modeling constructs suffices for describing all of the necessary feature information for large and complex product lines.

To capture features, here is the set of feature-modeling constructs (provided by Gears) that GM is using for its product line work. These constructs have evolved over 10 years based on experience in ever-larger and more complex industrial applications. The set of constructs has remained stable and small. They are:

- *Feature declarations* are parameters that express the diversity in the product line for a system or subsystem. Feature declarations typically express the customer-visible diversity among the products in a product line.

  Feature declarations have types. When a feature is chosen for inclusion in a product, it must be given a value consistent with its type. Table 15.1 shows the feature types supported by Gears.
- *Feature assertions* describe constraints and dependencies among the feature declarations. Feature assertions in Gears express REQUIRES or EXCLUDES

Table 15.1 Gears feature types

Boolean	True, false
Integer, Float	Signed or unsigned numeric value
String	Character string delimited by double quotes
Character	Single character delimited by single quotes
Enumeration	*Select exactly one* value from subordinate features
Set	*Select zero or more* values from subordinate features
Record	*Select all* values from subordinate features
Atom	Named member/value of a set or enumeration

relations. They express the constraint that a feature (or combination of features), if present, either requires or excludes the presence of another feature (or combination of features). For example, an assertion could express the need for software-actuated brakes to be present whenever the park assist option is on the vehicle or the need for certain switches to be present if certain lights are installed.

- *Feature profiles* are used to select and assign values to the feature declaration parameters for the purpose of instantiating a product. A feature profile is associated with a product and reflects the actual choices you make: Two door with sport package but no moon roof or four door with luxury package and moon roof. The values assigned in feature profiles must satisfy the constraints and dependencies expressed by the assertions in the feature declarations.
- *Assets* are the abstraction for systems and software artifacts in a production line. They are the building blocks of the products in the product line. Assets may be requirements, architecture and design documents, source code files, calibration sets, test cases, and so forth—artifacts from any phase of the development life cycle.
- *Variation points* encapsulate the variations in the assets used to build products. Feature declarations are mapped to these variation points, and a feature profile is mapped to the choices made at each variation point when building a product. In Gears, a variation point is instantiated from one or more variants, one of which will "stand in" for the variation point when a feature profile is used to build a product. A variant can "stand in" as is (in which case, the variation is accomplished by choosing which variant to use), or it can "stand in" after being transformed by applying a match-substitution pattern expressed in the regular-expression language of Perl. Also encapsulated within each variation point is the logic, expressed as a sequence of rules, that maps feature values to the different instantiations of that variation point.

There are three more Gears constructs that come into play in hierarchical product lines; these will be discussed shortly.

Figure 15.1 illustrates how this small set of constructs give us the concept of a *production line* (the part of the figure inside the red box). Assets are built and maintained on the left; each is endowed with one or more variation points (indicated by the gear symbol). Feature profiles determine how the assets are instantiated (by exercising their variation points) to produce product-ready artifacts. Under this paradigm, organizations become production-centric rather than product centric.

5.2 *Consistent Variation Management in Artifacts Across the Full Engineering Life Cycle*

The 2GPLE paradigm treats artifacts across the full engineering life cycle as equals, as current applications of product line engineering are demanding it.

It has long been a stated tenet of product line practice that core assets include more than software. For example, the Software Engineering Institute's Framework for Product Line Practice [14] states that "architecture, requirements specifications, testing-related artifacts, budgets, schedules, plans, and production infrastructure can all constitute core assets." However, a complete systems and software PLE lifecycle solution requires more than just a statement of eligibility. It requires consistent treatment of the artifacts' variation points under the production infrastructure, so that a full set of demonstrably consistent supporting artifacts can be systematically generated for each product. The alternative, trying to translate between the different representations and characterizations of features and variations across the boundaries between stages in the life cycle, is untenable.

To illustrate, imagine that a requirements engineering team has embraced a PLE requirements management technique based on tagging requirements in a requirements database with attributes that differentiate feature variations in requirements. Further, the design team has adopted a UML tool and has embraced inheritance as the mechanism for managing PLE design variations. The development team is using a FODA [7] feature model drawn in a graphical editor, plus #ifdefs, build flags and configuration management branches to manage implementation variations. Finally, the test team has adopted clone-and-own of test plan sections, stored in appropriately named file system directories to manage their PLE test plan variations. Now imagine what would be needed to create a complete PLE 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? Trying to translate between the different representations and characterizations of features and variations creates dissonance at the boundaries between stages in the life cycle.

To resolve this quagmire, a key aspect of 2GPLE is not just inclusion of non-software artifacts, but consistent and traceable treatment.

The artifacts to support this process include requirements, system architectures and designs, source code implementation, calibration parameters, test cases, and documentation. Some of the documentation could be for suppliers, who will provide some of the necessary software and hardware components. At a company such as GM, the long-term goal can be that all of these are endowed with variation points, which can be exercised to correspond to feature choices.

Common representation of variation points is key to achieving traceability from requirements to deployment. Traceability is of great concern for GM. Every requirement needs to 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 needs to be traceable forward to its implementation and vice versa. Each requirement needs to be traceable to one or more test cases that validate whether or not the requirement is satisfied in the final product. Managing all of these artifacts consistently, by tying their variations to features, is the key to achieving this.

5.3 CM That Maintains Assets, Not Products or Asset Instantiations

Configuration management (CM) for a product line must allow the rapid reconstruction of any product version that may have been built using various versions of the PLE assets and development/operating environment. This capability is essential for rapid response to and remediation of any anomalies that arise in the field.

The most important aspect of CM in 2GPLE is that the full superset of available PLE assets (and not the individual products or systems) are managed under CM. A new version of a product is not derived from a previous version of the same product, but from the shared superset of PLE assets themselves.

Contrast this to product-centric CM, illustrated in Fig. 15.2. Suppose a defect is discovered in Product B after it has been deployed, and the defect is traced back to product B's requirements. The Product B team fixes the defect and redeploys. But Product B's requirements might have been borrowed from Product A's requirements, and Product N's code might have been borrowed from (defective) product B's. By the time all of the potential dependencies have been run to ground to make sure the defect is eliminated from every place it might occur in n products, $n(n - 1)$ interactions have occurred, for an $O(n^2)$ complexity.

By contrast, using the scheme shown in Fig. 15.1, the requirements defect will be fixed in the asset, not the products. The affected products will be regenerated. This is an $O(n)$ proposition. All that is required to reconstruct any product version is to store the *temporal context* for that product version. A temporal context is a vector of assets and the version of each that was used to build a version of product.

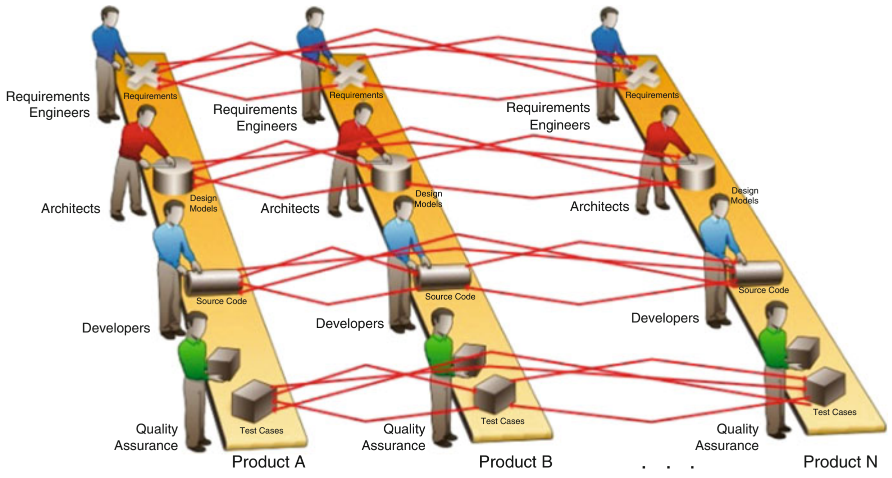


Fig. 15.2 A product-centric perspective with O(n^2) complexity

5.4 *Product Lines Across Organizational Boundaries*

For PLE to work at large organizations, it may be impractical to have a single organizational unit tasked with the care and feeding of the shared PLE assets [17]. Certainly having one global collection of feature declarations for an entire production line is impractical. (At GM, a single feature model for a car would comprise a few thousand features and have to be shared among thousands of engineers.) Further, subsystem engineers have no interest or need to see all of the feature diversity in other subsystems. For example, engineers for an automotive transmission system do not need to see feature abstractions that capture the diversity in the entertainment or GPS navigation system. It makes no sense to comingle them.

It makes much more sense to modularize the feature model in a way that corresponds to the organizational structure of the enterprise. Although these structures can change over time [5], they make an excellent starting point and let the organization begin to adopt PLE using familiar units.

At GM, a vehicle is composed from a set of integration areas (such as *safety* or *human–vehicle integration*), which assemble combinations of subsystems, which are in turn composed of functional elements, which are implemented by compositions of software components and calibrations that are loaded onto hardware components arranged in one or more physical architecture topologies. At each level in this decomposition—which is not necessarily hierarchical—engineers are assigned responsibility for managing the artifacts and configurations at that level, all of which are imbued with rich and numerous kinds of variation. Assembling a vehicle from the most primitive elements would simply be intractable. By contrast, a vehicle is more like a system of systems [10], which is managed as a product line of product lines. At GM the nesting is at least four levels deep.

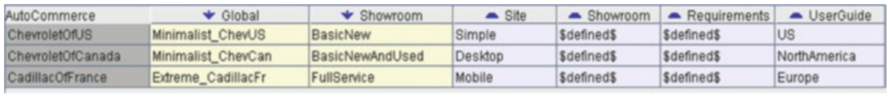
Each of these units represents a domain, by which we mean a body of knowledge [7]. Integration areas and subsystems are part of the fabric of the company. Building a subsystem for a vehicle, or combining subsystems in an integration area, or implementing a functional element requires specialized knowledge. In a PLE context, that specialized knowledge becomes knowledge about the variations that are possible, and the result is a number of product lines that each contribute instances to the overall vehicle product line.

Because some of the domains are quite large (as are the bodies of knowledge they embody), domains have sub-domains, and their product lines are the result of still finer grained nested product lines. (Section 8 will show an example of this nesting.) Features at the highest level of the hierarchy include things that vehicle customers would resonate with, such as daytime running lights or lane keep assist, while "features" take on a different meaning at the lower levels. Here, features represent a variation of a lower level "product" (such as a component that implements one of the available varieties of cruise control) being offered up to higher level product lines. But the whole chain starts at the highest level with the Bill-of-Features for the vehicle, each of which causes a cascade of lower level choices to be made. At every level, the same small and elegant set of concepts presented earlier work to capture the inherent variation. This lets engineers work largely independently within the confines of their own organizational units and domain expertise.

A hierarchical product line constitutes an architecture-like construct, in that there are interfaces and relationships among the nested product lines. There is the parent–child relationship for product lines that typically mirrors the system–subsystem decomposition in the vehicle architecture. Product line features can be partitioned, encapsulated, and scoped within the primary subsystems that realize the features. Features can also be shared among product lines by establishing an import relationship, which is crucial for establishing feature constraints and asset variation points among interrelated subsystems (e.g., a high-end flavor of cruise control that slows the car if there's traffic ahead requires a flavor of the braking system that supports braking via software command).

Gears provides three more constructs that facilitate the interfacing and coordination between levels in the hierarchical product line: mixins, matrices, and imported production lines.

- *Mixins*. Although many feature declarations will fall cleanly into the realm of one asset or another, there are many cases where a feature declaration applies to two or more assets. For example, the automotive platform (Buick Regal? Chevy Cruze? Cadillac CTS?) and the region for which the vehicle is being marketed (North America? Brazil? China?) constitute features that determine how an asset should be configured at many levels: Integration area, subsystem, functional element, component. Rather than duplicating the same feature declaration in multiple assets, Gears provides the mixin abstraction to allow creation of a feature declaration in one place and then "mix it into" the feature declarations of multiple assets, by reference.



AutoCommerce	▼ Global	▼ Showroom	▲ Site	▲ Showroom	▲ Requirements	▲ UserGuide
ChevroletOfUS	Minimalist_ChevUS	BasicNew	Simple	$defined$	$defined$	US
ChevroletOfCanada	Minimalist_ChevCan	BasicNewAndUsed	Desktop	$defined$	$defined$	NorthAmerica
CadillacOfFrance	Extreme_CadillacFr	FullService	Mobile	$defined$	$defined$	Europe

Fig. 15.3 A Gears matrix, with three rows for three products. The "Global" and "Showroom" columns show feature profile choices for mixins; the last four columns show feature profile choices for assets.

Mixins are more than a convenience to avoid typing of feature declarations. They also encapsulate, in a single location, the feature profiles for the feature declaration parameters. Having a single location for the feature profiles prevents inconsistencies when composing assets to create a complete system.

Imagine, for example, that we need the lane-keep-assist option to be supported by two assets. If this option were declared independently in both assets, it would be possible to inadvertently create a system where one asset assumed that the feature was supported and the other asset assumed it was not supported. Using mixins, there is a single feature profile for the lane-keep-assist option that is "mixed into" both assets. It is either true for both assets or false for both.

- *Matrices*. A Gears production line is the "virtual factory" that knows how to build products by configuring assets in accordance with selected feature profiles. To build a product, you need to tell Gears what feature profile to use for each asset and each mixin in the production line. A matrix is a table showing the choices to build a complete and consistent product. Each row specifies one product. Each column specifies a choice of feature profile for a mixin or an asset (Fig. 15.3).

  A complete product instance is "actuated" by actuating every asset and nested production line column that has an entry for that product. Each asset and nested production line is actuated according to its cell value in the row. If an asset imports a mixin, the mixin feature profile to be used is determined by its cell value in that row.

  Some products may not need all of the assets. For example, low-end products in a production line may not include "luxury" assets that are aimed at high-end products. Each matrix allows you to include or exclude individual mixins and assets to accommodate such product diversity.
- *Imported production lines*. Gears allows you to create a hierarchy of production lines by nesting one production line into another production line. In order to use a production line as a nested production line, it must first be imported. An imported production line will be added as a column in the matrices for the importing production line, just like an asset or mixin. For example, engineers at GM have defined a production line for the safety integration area. In order to provide a safety package to a vehicle, the safety production line must include specifically configured subsystems from a number of subsystems (such as body and active safety), which are their own production lines. A subsystem production line, in turn, can import production lines corresponding to functional elements, and so forth.

Hierarchical product lines are not new in the product line literature [2], but GM is turning out to be the largest (and deepest) realization seen in practice. Also, hierarchical product lines are not needed just because the product line is large. We know of another product line of multi-million-line systems with equally astronomical product variation. However, their products are structured as a set of a dozen or so major subsystems with limited influence on each other in terms of variation selection. That is, choosing features for one subsystem does not have much influence on the features of the other subsystems, obviating the need for a hierarchical production line.

5.5 Industrial-Strength Automation

The last ingredient in 2GPLE is a configurator employed to maintain configurations and translate feature profiles into assets with their variation points exercised in prescribed ways. The tooling needs to be able to support the construction and management of feature models (including feature declarations, assertions, and profiles), assets and their variation points, support hierarchical production lines, and map from feature choices to asset instances (in Gears, this is the job of the matrices). Further, it needs to either provide version control for the models and artifacts or (even better) work seamlessly on top of the user's own choice of change management system.

A major requirement for the tooling is that it supports the specification and selection of variation in assets and artifacts from across the entire spectrum of the product life cycle. This means that in addition to working with open-format artifacts, the tool will have to support variation proprietary-format artifacts such as IBM Rational DOORS requirements modules, Microsoft Word documents, and Excel spreadsheets, build files for Make or Ant, Rhapsody UML models, and many more.

Gears supports these and more using *bridges*. A bridge is a piece of software that "knows" the other-tool representation and presents a "product-line-aware" user interface for that tool that allows product line engineers to insert variation points in the artifacts maintained by that tool.

First-generation approaches always discussed the need for automation; second-generation approaches require it, and it must interface with other system and software engineering tools.

6 A Mega-scale Product Line at GM

General Motors is the largest automotive manufacturer in the world [1]. In 2011 it sold over nine million vehicles, produced (with its partners) in 31 countries around the world. That works out to over 1,000 vehicles rolling off assembly lines *every hour*.

Fig. 15.4 Chevy Volt: Ten million lines of code, nicely packaged (© GM Company)

The product line we describe is built under the Next-Generation Tools (NGT) initiative at General Motors. GM introduced NGT to tackle the complexity brought on by (among other things) the introduction of hybrid and alternative-fuel vehicles and new “active safety” features that require intricate and unprecedented orchestration among vehicle subsystems. Product line engineering is a key ingredient of NGT.

General Motors may well represent the most challenging domain in all of product line engineering. We characterize it as *mega-scale PLE* due to the fact that engineers must deal with multiple product line characteristics that measure in the millions although, as we will see, even this term’s implied order of magnitude fails by a wide margin to do justice to the problem space:

- *The vehicles are complex*. As a group, GM vehicles comprise some 300 engineered subsystems such as brakes, exterior lighting, interior lighting, entry controls, and many more. The Chevrolet Volt runs approximately ten million lines of code [12], which is several million more than either the Boeing 787 or the F-35 Joint Strike Fighter 13 (Fig. 15.4).
- *The variation among vehicles is enormous*. GM builds over 60 models under seven brands and divisions. The vehicles may be internal combustion, electric, or both. Customer-visible options include everything from power windows to “lane keep assist” (a system to help the car stay in the correct highway lane). These options, and many dozens more, fundamentally affect the electronics and software aboard the vehicle.

  Legislation, not to mention cultural preferences, in the 150+ countries where GM does business also imposes feature constraints. To choose one of many dozens of examples, there are complex interactions between the vehicle’s exterior lights (low beam headlights, high beam headlights, tail lamps, brake lights, parking lights, daytime running lights, front fog lamps, rear fog lamps, cornering lamps, reversing lamps, and hazard flashers) in terms of which lights are allowed, disallowed, or required to come on with which others. The “lead me to my car” feature makes lights come on or flash when the driver presses a button on the key fob. Which lights come on, whether they flash or not, and how long they stay on all are specific to the region and (of course) what exterior lights are actually on the vehicle. The electronics aboard *every* car has to get that behavior right for *that* car.

A simple thought experiment helps to grasp the astronomical magnitude of the variation involved. We can think of vehicle rolling off an assembly line as the result of making a very large set of yes-or-no decisions. The set of all possible vehicles results from all possible combinations of those yes-or-no choices. The size of that product space is 2^x, where x is the number of decisions. If $x > 260$, then the product space comprises more combinations than the number of atoms in the observable universe 21. For GM, x is in the low thousands. (The number of variants that GM actually produces is much less than that, obviously—a number in the low tens of thousands.)

- *Feature interaction abounds*. The lighting example above illustrates interactions within a subsystem (exterior lighting), but other features require complex interactions among completely different subsystems. For example, the presence of "park assist" (a feature to help park the car) requires the presence of a sensor to gauge the car's position relative to the parking space. In some cars this will be a sonar detector, while on others it will be a camera. Park assist also requires brakes that accept software control, and some versions of park assist require particular versions of steering controls. Thus, the presence of a customer-visible feature can affect multiple subsystems, requiring communication and coordination among the subsystems on the car, and among the groups that are responsible for the subsystems involved.
- *The product line must be in lockstep with current and future model years*. GM has to plan their production years in advance. Features that won't be in the showroom for 3–5 years are already part of today's engineering. And the entire product line marches in unwavering lockstep with the calendar, fixed and unforgiving, which defines each new model year. This means that the product line infrastructure must support concurrent engineering streams for each of the fixed yearly cadences, as well as concurrent development cadences for release cycles scheduled every 6 weeks throughout the year. There may be as many as 15 active, concurrent engineering baselines that engineers must contribute to and coordinate among.

  Another thought experiment illustrates the astronomical combinatorics of the temporal dimension. Each of the 300 or so GM subsystems will typically undergo enhancements or fixes within ten or more cadences within a 2-year period, resulting in 10^{300} possible subsystem version combinations. As with the number of feature combinations, this also vastly exceeds the 10^{80} atoms in the observable universe [18].
- *Consistency and traceability across the life cycle are required*. Each vehicle is the result of an engineering process that spans requirements, design, implementation, calibration, layout and interconnection of electronic control units (ECUs), allocation of software to the ECU network, production of a manufacturing bill-of-materials, and testing. Each of the artifacts must be consistent with each other, in that they must all be accurate with respect to the vehicle to which they

apply. Further, that consistency must be demonstrable through feature interdependency constraints as well as traceability among lifecycle phases.

- *The organization is very large*. Ultimately up to 5,000 engineers will be directly working on artifacts that are part of the product line, some in roles newly defined expressly to support the PLE effort.

The emergence of hybrid and alternative fuel vehicles and new active safety features, which dramatically increase the amount of product line diversity, plus the new economic reality in the automotive industry that leaves little margin for technical error, drove GM to plan to overhaul its engineering tools and processes. The result is the Next-Generation Tools (NGT) initiative.

7 GM's Approach for Mega-scale PLE

This section describes in greater detail how GM has adopted 2GPLE as their technical roadmap for the future.

7.1 GM's Architectural Decomposition

GM's architectural strategy plays a key role in how it is rolling out PLE. The strategy is one of logical decomposition as a way to gain control over the complexity of building a vehicle's electronics and a way to allot the thousands of engineers into organizational units with clearly scoped roles and responsibilities.

- *Functional architecture*. First, a vehicle consists of a number of *domains*. These are "containers" for capturing the requirements necessary to describe the electronics terms applicable to an entire vehicle. Domains define areas of related functionality. For example, Powertrain is a domain, as is HVAC (heating, ventilation, and air conditioning).

  Orthogonal to domains are *integration areas*. Integration areas can be thought of knowledge areas for satisfying high-level stakeholder requirements for vehicles. Requirements here span domains. For example, Noise and Vibration is an integration area; it "touches" any domain that can contribute noise or vibration to the occupants' driving experience: Powertrain, Body, Chassis, HVAC, and more.

  GM refers to integration areas and domains together as its *functional architecture*. The functional architecture provides the overarching structure to host the hierarchical PLE models. Each domain or integration area team will build the PLE models for their area of concern in corresponding part of the functional architecture hierarchy. Figure 15.5 illustrates.
- *Implementation architecture*. Domains comprise *subsystems*. Subsystems represent physical systems on vehicles. There are subsystems for brakes, external

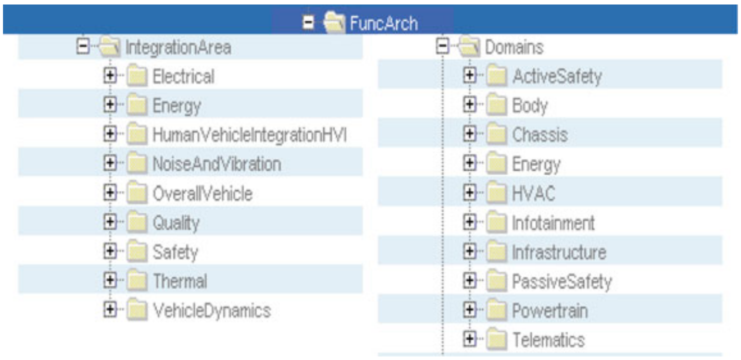


Fig. 15.5 Tool view of GM's functional architecture, showing some of the integration areas and domains

lighting, internal lighting, entry and egress, and many more. Subsystems have their own requirements, which must permit the subsystems to play their proper role in the domains and (in turn) integration areas that need them. Subsystem designers in turn decompose their subsystems into *functions*, and functions into *functional elements*, and write requirements for each. *Components* are units of implementation that satisfy the requirements for functions and functional elements. Components are arranged in a decomposition hierarchy; leaf nodes are components; higher nodes (which are just aggregations of their descendants in the tree) are called *compositions*. Components may be software components or hardware components, depending on how the functional elements are satisfied. GM calls this component structure (with components mapped to the functional elements they satisfy) its *implementation architecture*.

- *Deployment architecture*. Next, the components have to be assigned a place in the onboard electronic architecture topology. Software components need to be assigned to an electronic control unit (ECU), and hardware components have to be assigned a spot in the topology. The selection of a topology from a small number available, the assignment of ECUs to spots in the topology, and the assignment of software to ECUs all constitute what GM calls its *deployment architecture*.
- *Vehicle application architecture*. Finally, the components need to be laid out on a vehicle. This architecture determines where the ECUs are stationed, and the type, position, and routing of the wire harnesses to connect the sensors, actuators, and ECUs.

These architectures—functional, implementation, deployment, and vehicle application—institutionalize and add structure to concepts that are deeply ingrained in the organizational and technical fabric at GM. For instance, there are centers of deep expertise in brakes and lighting and keyed/keyless entry systems and dozens of other domains. As part of PLE adoption, these centers are not going to be discarded in favor of a massive reorganization involving the reorientation, reassignment, and retraining of some 4,000–5,000 engineers.

In order to maintain the functional architecture and its rich set of decomposition structures, the hierarchical production line approach of 2GPLE is needed. Integration areas, domains, subsystems, and functional elements can all be represented with their own production line, importing the production line of smaller, child units. Software components and ECUs can be represented as assets within the production line at the appropriate decomposition level. Across all levels, requirements, design models, and specifications can also be represented as assets. The result is a small and nondisruptive change from the organizational schemes that GM has employed successfully for years.

7.2 Roles

GM's embrace of 2GPLE has led to the creation of a few new and refined engineering roles that have come about as a direct result of piloting their hierarchical product line. The major PLE roles and their broad responsibilities vis-a-vis maintaining the PLE models and artifacts include:

- *Feature Owner*. Feature owners take ownership of GM features (customer-visible features such as cruise control or lane keep assist or hundreds of others that are visible and bring value to car owners). These features are, in a sense, abstractions. They only become tangible when realized by concrete artifacts: requirements, functions, software components, electronic control units, and wiring. In GM's PLE environment, each of those artifacts will also embody variation. It is the feature owner's job to make sure that all of those artifacts in "supporting roles" are adequate and correctly provide the feature to GM's customers.

  A feature owner is the main technical contact to external teams who need to know about the feature from the point of view of assembling a vehicle from this and other features. This engineer is the recognized expert for the functional area regarding the feature's required variants, the system constraints it imposes, and how it integrates onto a vehicle platform.
- *Functional Architect*. This engineer owns a specific area of the functional architecture and as such establishes ownership and boundaries between system-level assets. Together, the functional architects maintain the functional architecture taxonomy introduced above.

  With the advent of the NGT 2GPLE effort, functional architects have taken on a new and critical role. Together, they are the keepers and centralized owners of all of the PLE models. Their job is to ensure that the PLE models produced inside their assigned area by feature owners, asset owners, and others are consistent, fit together, and represent best PLE practice.

Functional architects are each assigned a domain, which will involve several subsystems. As PLE practices are introduced into each domain, functional architects will actually build the PLE models, working with feature owners who are the subject matter experts in each area. For example, the functional architects will mine the feature owners' knowledge about what constitute the features in an area, what profiles (choices of feature combinations) they should provide, and what feature combinations require or exclude other feature combinations. For example, the feature owner for the wipers and washers knows that rear wipers cannot be installed on any vehicle with a rear window that slides open; the functional architect will capture this with an "EXCLUDES" assertion between the rear window type and the wiper/washer configuration.

Under this scheme, the feature owners remain the subject matter experts about their features; the functional architects translate (or help the feature owners translate) that knowledge into well-structured and consistent PLE models.

The PLE models built with Gears for each domain or subsystem take the form of production lines that are then combined by importing them into an overarching production line for the entire vehicle, making full use of the cross-organizational, hierarchical product line aspect of 2GPLE.

- *Product Line Integration Engineer*. This is another new role at GM, brought about by PLE. This engineer collaborates with Vehicle Product Teams in the selection of a "bill-of-features" for a vehicle being planned. The product line integration engineer also collaborates with the feature owners in the identification of the top-level subsystem production line "products" that will be offered up to vehicles. The vehicle team for a vehicle will need the services and advice of a team of product line integration engineers, who together will put together the bill-of-features for that vehicle's electronics system. When the bill-of-features for a vehicle is created, the product line integration engineers will be at the table.

  For example, the vehicle team for the Buick Verano wants to understand what kind of climate control options they can offer with the car (or, to put it another way, what climate control features are eligible for the Verano's bill-of-features, and what the downstream implications are of each). The product line integration engineer responsible for heating, ventilation, and air conditioning (HVAC) systems will offer up various automatic and manual climate control systems. If a vehicle might 1 day be powered by hybrid or next-generation energy and propulsion systems, this might mandate another kind of HVAC system.

  The vehicle teams aren't interested in the details of the features' implementations, but only in how the features will appear to the customer and how they interact with each other. The product line integration engineers, then, manage subsystem "products" that are exposed at the vehicle and bill-of-features level.
- *Asset Owner*. These engineers manage various kinds of assets across their life cycle, and establish variation points in the assets. A requirements engineer is one kind of asset owner. Their responsibilities include migrating requirements from legacy requirements assets (mostly Word documents) into DOORS and, along the way, imbuing those requirements with variation points that support features.

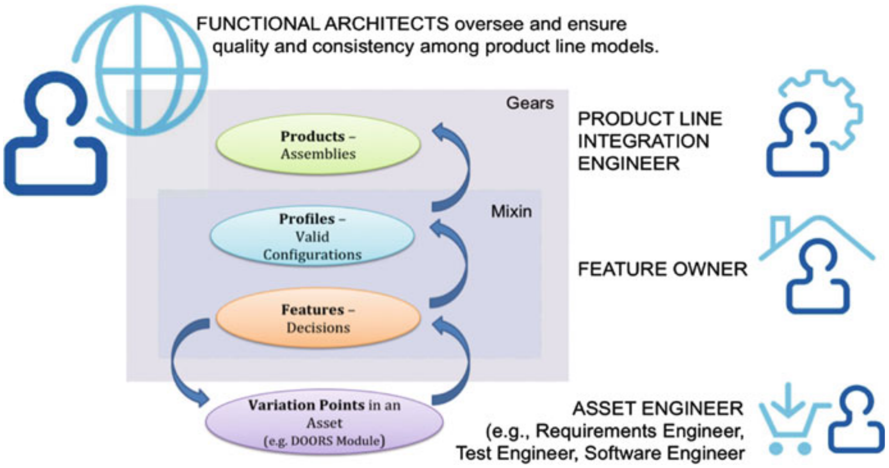

FUNCTIONAL ARCHITECTS oversee and ensure quality and consistency among product line models.
Gears
Products – Assemblies
Mixin
Profiles – Valid Configurations
Features – Decisions
Variation Points in an Asset (e.g. DOORS Module)
PRODUCT LINE INTEGRATION ENGINEER
FEATURE OWNER
ASSET ENGINEER (e.g., Requirements Engineer, Test Engineer, Software Engineer

Fig. 15.6 PLE constructs and roles at GM

Asset owners, including requirements engineers, are responsible for modeling the features that their assets make available to the consumers of those assets, and the variations in those features. These features are often strongly suggested, if not identified outright, in existing technical specifications. Thus, feature creation is more of an identification and extraction process, as opposed to an invention process. This helps things go more smoothly and predictably.

Figure 15.6 shows how these roles relate to the PLE concepts discussed previously.

7.3 *Organizational Adoption*

Launching and institutionalizing [15] this approach at GM has required significant investment over the last 2 years or more, and that investment is ongoing. There has been a group of champions and advocates of the PLE approach throughout the effort. Early on, they sponsored a 2-week workshop to show how the approach (using Gears in concert with DOORS) could tame the requirements for a major subsystem, with variations clearly identified and managed. This pilot effort produced more strong advocates and steered GM towards their current tooling approach.

After that followed a steady series of workshops and technical meetings with senior engineers to work out how to apply the concepts at GM; the eventual results of these meetings include the architectures and roles described above, plus a vision of how features could be used across all of the architectures to describe variation. All the while, the champions practiced internal evangelizing, advocating the approach to management and engineers alike. One-day requirement workshops were held with subsystem owners to duplicate the results of the first 2-week workshop.

The latter part of 2011 saw the launch of a series of some two dozen *Bill-of-Features Workshops*. These workshops bring together a small group of feature owners and subsystem experts in a particular area—for example, interior lighting. They spend a day learning the PLE approach and then actually using Gears to model the features in their domain. An important goal is to have participants experience the "PLE epiphany," when they see how 2GPLE and the NGT tool suite will help them do their jobs better.

At the start of 2012, after 2 years of establishing buy-in, GM launched a series of training courses. The course series kicks off with a short introduction to PLE at GM and continues with 1-day classroom courses in each of the tools and how they will be used. In concert with the training is the establishment of resources to help engineers once they go back to their desks: Discussion boards, FAQ lists, help desks, and the like. The Bill-of-Features Workshops have continued, with the features created for the domains and subsystems being used to provide a full pallet of vehicle-level variation choices, plus create variation points in assets such as requirements specifications. Thus, the features and profiles in the middle of Fig. 15.6 are being used to inform the product (vehicle) assemblies at the top and the requirements assets at the bottom. The PLE roles identified earlier are all carrying out their respective responsibilities using features as the *lingua franca* for what they do.

Answers to recurring questions are being captured and stored in a "GM PLE Cookbook," which will include a set of patterns and anti-patterns for good practice, a list of FAQs, and a set of naming conventions for product line objects shared across organizations. This will represent a trove of practical knowledge not usually divulged in the product line literature, as well as another aid to institutionalization at GM.

7.4 What Is the End Game?

One of GM's senior electronics engineers characterizes the electronics division's job this way: "We build silver boxes," he said, "load software in them, and wire them together." If they can do that correctly for every vehicle they build, their job is done.

Whatever PLE and NGT can do to help achieve that purpose is a win for GM. The long-term vision is to create a bill-of-features for a vehicle (which manifests as a vehicle-level feature profile in Gears) and automatically derive as much as feasible of the bill-of-materials for that vehicle, including requirements, designs, models, software, calibration data, tests, documentation, allocation of software to hardware, wiring diagrams, and so forth. That vision is years away from being achieved.

However, short of that, there are some intermediate steps that GM is working towards. Examples include

- Migrating all requirements to incorporate Gears variation points that formalize feature-based variations in the system, subsystem, and component requirements
- Generating calibration files and values that will need to be loaded onto the electronics modules or
- Given a set of features on a car and the components that need to be onboard to support those features, generating a list of all of the digital signals that the serial networks will have to accommodate

Longer term goals include calculating certain additive nonfunctional properties of the electronics, such as weight or generated heat or cost.

Even short of this capability, GM is already getting value out of their PLE efforts even before they have started producing instantiated engineering artifacts. Just defining an internally consistent vehicle with consistent versions of subsystems, functional elements, components, and hardware allocations represents a very big step in managing the complexity at hand. To be able to do this in an end-to-end fashion under the auspices of fully interoperating tool suite is a capability not available at GM before now. The automation—in this case, Gears—can do a semantic check on the feature model and report anomalies, such as the fact that this vehicle is supposed to support the lane-keep-assist feature but the instrument cluster chosen for it doesn't have the correct display for that feature, or the chosen physical architecture topology cannot support the serial data communication required.

8 Example: Daytime Running Lights

We conclude by illustrating some of the points in this chapter through an example, which necessarily must be a small one. A *daytime running light* (DRL) is a "lighting device on the front of a roadgoing motor vehicle, installed in pairs, automatically switched on when the vehicle is moving forward, emitting white, yellow, or amber light to increase the conspicuity of the vehicle during daylight conditions" [20].

8.1 DRL Requirements

DRLs are considered a feature at GM; they're certainly visible to the user. But not all cars have them. DRLs are required equipment in Canada, Norway, and Sweden, prohibited in Japan and China and optional in the USA, Europe, Australia, and the rest of the world. (Region of sale turns out to be a major discriminator among features, permitting or precluding a plethora of other features.)

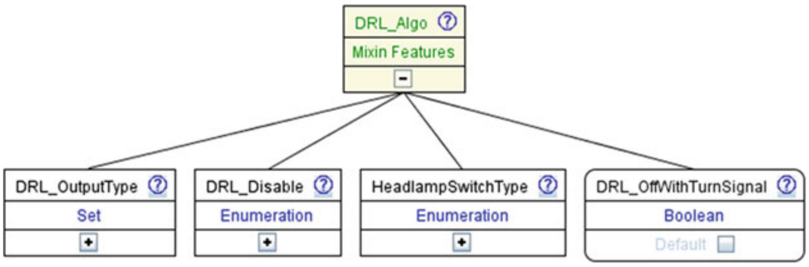


Fig. 15.7 Feature model for daytime running lights

In vehicles that have them, DRLs can be "implemented" by lamps dedicated to that purpose, or by front turn signal lamps, reduced intensity low beam headlamps, full intensity low beam headlamps, parking lamps, or a combination of parking lamps and dedicated lamps.

Just as there are many ways to realize DRLs, there are many choices for how the customer can turn them on and off (including none at all, leaving it up to the car to do so automatically). There is a thicket of requirements concerning when DRLs may, must, and may not be on. For example, in Europe, DRLs must switch off automatically when the front fog lamps (if the car has front fog lamps) or headlamps are switched on, except when the headlamps are "used to give intermittent luminous warnings at short intervals"—that is, flashed.

These and other impinging factors consume page after page in the requirements document for the exterior lighting subsystem, of which DRLs are a member. These requirements are rife with information about what requirements apply under what conditions and be used to identify variations in the DRL feature.

Besides being a feature by themselves, DRLs play a part in other features as well. Some realizations of the "Lead me to my car" feature flash the DRLs when the key fob is pressed. Police vehicles have turn-everything-on features, which include dedicated DRLs if the car is so equipped. Cornering lamps, another feature, can only come on under certain conditions and affect DRLs if they share output devices.

8.2 Modeling DRLs

The feature owner for DRLs is responsible for understanding how the DRL feature is realized, the variations it includes, and any variant capabilities required because of its appearance in other features.

Figure 15.7 shows a preliminary feature model for DRLs. The feature model captures the output type (what lamps on the vehicle can be used), if and how DRLs can be disabled, and how DRLs are integrated with the car's headlamp controls turn signals, respectively.

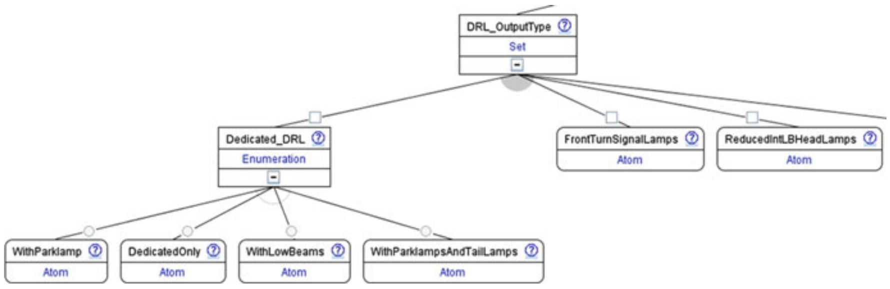


Fig. 15.8 Partial expansion of the OutputType feature of DRLs

Figure 15.8 shows how the output type subfeature is expanded to take into account all the possibilities. The output type is modeled as a set; a vehicle can have any number of ways of realizing the DRL feature, or none at all.

However, if a vehicle realizes the DRL feature with low-intensity headlamps, then it cannot realize it with high-intensity headlamps as well and vice versa. An assertion captures this:

NOT `DRL_Algo.DRL_OutputType` >= `{ReducedIntLBHeadLamps, FullIntLBHeadLamps}`

This says that the OutputType set cannot include both of the indicated values; in Gears, the symbol ">=" (when used in an expression involving sets) means "is a superset of."

The DRL feature owner builds these feature models in Gears, under the conventions and standards developed by the functional architects, and in particular the functional architect for the exterior lighting domain. He or she will also build a matrix for the DRL feature model that specifies a small number of flavors of the DRL feature that can be made available to Product Line Integration Engineers working to assemble vehicles from features.

Simultaneously, requirement engineers who own the requirements for this feature work to turn the DRL requirements into a Gears asset, with variation points that (when exercised) produce requirements that correspond to the requirements needed for each case.

8.3 DRLs and Deployment

Those responsible for choosing a deployment architecture are another kind of asset owner; their asset is the set of ECUs needed for the features on board, and the variation points they can provide based on features and feature combinations chosen. These variation points include basic network topology (currently two are available; more may be added), how many ECUs will populate a topology, what the choice of ECU hardware will be, and the allocation of software components to each ECU.

8.4 DRLs and Other Features

Feature owners for other features that interact with DRLs (such as the lead-me-to-my-car feature owner) will need to reference DRLs in their feature models and profiles. They will coordinate with the DRL feature owner, under the auspices of the functional architects for the including domain or domains, to make sure that the DRL feature can be referenced, by importing the DRL domain-level mixin into their domain production line.

Other domains than exterior lighting will need the same ways to refer to DRL in their feature models. For instance, the switches that turn DRLs on and off are part of the Body domain, whereas any indication that DRLs are on are part of the Displays domain.

8.5 DRLs and the Vehicle

Finally, those defining a vehicle type and the myriad of features combinations that GM wishes to offer with it, can do so by importing all of the domains' and integration areas' production lines, adding the highest layer to the product line hierarchy. They will also define a matrix of "products" for each vehicle, defining combinations of features in concert with each other.

9 Outlook

The guiding PLE vision at GM is the ability to engineer vehicles—across the full life cycle—according to a "bill-of-features" rather than the traditional "bill-of-materials." Although still very much a work in progress, the GM experience has already revealed a number of lessons about mega-scale product line engineering.

First, the product line experience at General Motors can be seen as intensively applying aspects of what has been called Second-Generation Product Line Engineering. This new perspective brings the following ideas, which previous approaches always allowed but never stressed, to the forefront:

- A focus on features as the "lingua franca" of variation and product selection; the "bill-of-features" replaces the "bill-of-materials" as the key engineering artifact for product derivation. At GM, functional architects and feature owners cooperate to capture the features in Gears models across domains and subsystems and integration areas. Vehicle-level engineers can choose from the profiles provided from across the functional architecture, and asset owners design and install variation points in their assets that are expressed in terms of those very same features.

- Treatment of artifacts across the entire life cycle completely consistently with each other, and consistently with the software, as first-class components of the product line and the derived products. At GM, requirements and calibration sets are the near-term artifact targets, with, wiring data and source code components on the horizon.
- An emphasis on high-quality automation at the center of a production line, to quickly turn a bill-of-features into a set of instantiated lifecycle assets. At GM, this automation takes the form of the Gears configurator, working at all levels of the functional architecture and in separate groups, from vehicle-level engineers, down through domains and subsystems, as well as assets.
- A CM and PLE approach geared to multi-baseline multiproduct management in a way to reduce the order of complexity from $O(n^2)$ to $O(n)$. GM has embraced this configuration management paradigm by only managing the shared assets and not their auto-configured instances.
- Taking multi-organizational management in stride, by providing feature model concepts such as mixins and imported (hierarchical) production lines, to reflect the structure of engineering activities and domain knowledge present in an ultra-large organization. This is perhaps the most overarching aspect of the GM story. PLE would not have worked at GM by overthrowing their longstanding multi-level functional architecture and corresponding organizational hierarchy. Such a radical departure from their current way of thinking about and organization to build vehicles would probably have ruled out any attempt at a large-scale PLE effort; the organizational change would have been too forbidding. Instead, they are able to apply PLE at every level and in every group of their functional architecture and make their PLE models work together using the Gears constructs of mixins, matrices, and imported production lines.

GM's PLE approach embodies a compelling need for each one of these characteristics. They have embraced feature-based variation at all levels of their product line to the extent that they are transitioning from an organization dominated by subsystem owners to one where *feature owners* play the key role.

Second, the GM experience also validates that a small set of consistent concepts is sufficient to model product lines of inordinate complexity. Features (declarations, types, assertions, and profiles), assets (that embody features, as well as variation points), mixins, and matrices constitute a production line, the "factory" that turns feature choices into asset instances. Allowing production lines to import other production lines gives us unlimited hierarchy, which can map to any organizational structure in which specialized bodies of knowledge are encapsulated throughout, no matter how many levels deep.

Third, one of the most important aspects of PLE and GM is the application of consistent variation management in artifacts from all across the life cycle (the second bullet above). In order to accomplish this, the automation engine has to embody business partnerships with important tool vendors.

Future work involves continuing the march towards the ultimate "end game": Generating a complete bill-of-materials for a vehicle by starting with its bill-of-features.

Ultimately, GM may investigate merging PLE with product lifecycle management (PLM), which is the technology used in vehicle manufacturing. That would represent a convergence with repercussions across the entire manufacturing industry.

Acknowledgments Our thanks to everyone at General Motors who is involved in making their adoption of 2GPLE a success, and especially those who have been involved through the roll-out effort. Their contributions have made this case study, and the story it tells, possible.

References

1. G.M. regains the top spot in global automaking. Business Day, New York Times, 19 Jan 2012
2. Bosch, J.: Organizing for software product lines. In: Proceedings of the 3rd International Workshop on Software Architectures for Product Families (IWSAPF-3). Las Palmas de Gran Canaria, Spain, 15–17 Mar 2000, pp. 117–134. Springer, Berlin (2000)
3. Catalog of Software Product Lines. Software Engineering Institute, http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm
4. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading, MA (2002)
5. Clements, P., Brownsword, L.: A case study in successful product line development. Software Engineering Institute CMU/SEI-96-TR-016, September 1996
6. Cohen, A.: Russia trails U.S. in pursuit of a fifth-generation jet. UPI, 14 Jan 2009, United Press International, retrieved 2012: http://www.upi.com/Business_News/Security-Industry/2009/01/14/Russia-trails-US-in-pursuit-of-a-fifth-generation-jet/UPI-35761231951126/
7. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021, ADA235785). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
8. Krueger, C.: BigLever software gears and the 3-tiered SPL methodology. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA '07), pp. 844–845. ACM, New York (2007). doi:10.1145/1297846.1297918, http://doi.acm.org/10.1145/1297846.1297918
9. van der Linden, F.J., Schmid, K., Rommes, E.: Software Product Lines in Action. Springer, New York (2007)
10. Office of the Deputy Under Secretary of Defense for Acquisition and Technology. Systems and Software Engineering. Systems Engineering Guide for Systems of Systems, Version 1.0. ODUSD(A&T)SSE, Washington, DC. http://www.acq.osd.mil/sse/docs/SE-Guide-for-SoS.pdf (2008)
11. Parnas, D.L.: On the design and development of program families. IEEE Transactions of Software Engineering **SE-2**(1), 1–9, March 1976
12. Paur, J.: Chevy Volt: King of (Software Cars). Wired, 5 Nov 2010, http://www.wired.com/autopia/2010/11/chevy-volt-king-of-software-cars/
13. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (1998)
14. Software Engineering Institute. A Framework for Software Product Line Practice, version 5.0. http://www.sei.cmu.edu/productlines/frame_report/index.html
15. Software Engineering Institute. A Framework for Software Product Line Practice, version 5.0: Launching and Institutionalizing. http://www.sei.cmu.edu/productlines/frame_report/launch.inst.PL.htm
16. SPLC Software Product Line Hall of Fame. http://splc.net/fame/gm.html
17. Van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action. Springer, Heidelberg (2007). Chapter 5

18. Villanueva, J.C.: Atoms in the Universe. The number of atoms in the observable universe is approximately 10^{80} or 2^{260}. http://www.universetoday.com/36302/atoms-in-the-universe (2009)
19. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, Reading, MA (1999)
20. Wikipedia. Daytime running lamp. http://en.wikipedia.org/wiki/Daytime_running_lamp
21. Wikipedia. Fifth-generation programming language. http://en.wikipedia.org/wiki/Fifth-generation_programming_language