To be presented at 26th Annual INCOSE International Symposium (IS 2016)
Edinburgh, Scotland, UK, July 18-21, 2016

It Takes a Village: Why PLE Technology Solutions Require Ecosystems of PLE Technology Providers

William J. Bolander
IBM
william.bolander@us.ibm.com

Paul C. Clements
Charles Krueger
BigLever Software
pclements@biglever.com
{pclements, ckrueger}@biglever.com

Copyright © 2016 by IBM and BigLever Software. Published and used by INCOSE with permission.

Abstract. In product lines with many products and demanding production tempos, manual product derivation is simply not feasible, which leaves us with the option of automated product derivation. That is easily enough said, but what precisely is the role of that automation? While commercially available PLE product production tools are available (usually called *configurators*), they cannot by themselves provide a complete engineering solution to a product line organization. Instead, an *ecosystem* of tooling is required. This paper explores what capabilities a PLE tool needs to have, why other tools are necessary as part of a holistic PLE technology solution, and what roles that other tooling must play.

Introduction

Modern large-scale industrial systems and software product lines are continuing to emerge that render manual product production (sometimes called product derivation) strategies [4] ineffective. The number of products can be too large (a large automotive company might produce tens of thousands of variants in its product line, for example [9]) – or the tempo of product release can be too fast (a software-focused company like HomeAway might release products many times a week [15]), to allow for labor-intensive practices to turn product line shared assets into products. If we accept that self-evident premise, then obviously we must turn to *automated* product derivation in this arena.

But what technology or technologies are available and required to fulfill that need? The answer is more complex than simply running out to purchase your favorite product line engineering (PLE) tool. In fact, the technology or technologies that product companies *already* bring to bear on their products, whether developed as a product line or not, must be taken into consideration. So must technologies that help the product line as a whole evolve systematically and sustainably over time. These and other considerations lead us to realize that a PLE technology *ecosystem* is required for large-scale industrial strength product line development.

This paper explores why ecosystems are needed and what the components a full-fledged PLE ecosystem might comprise. We conclude by offering observations on what these observations mean to technology providers who hope to succeed in the large-scale PLE arena, and PLE practitioners who are looking for a holistic technology solution to power their product line engineering efforts.

Essentials of product line engineering (PLE)

Systems and software product line engineering, abbreviated here as *product line engineering (PLE)*, refers to the disciplined engineering of a portfolio of related products using a common set of shared assets and a common means of production.

Products

The products in the portfolio are described by the properties they have in common with each other and the variations that set them apart. The products can comprise any combination of

- software
- systems in which software runs, or
- non-software systems that have software-representable artifacts (such as engineering models or development plans) associated with them.

Herein, when we refer to a product we usually mean not only the primary entity being built and delivered, but also all of the artifacts that are produced along with it. Some of these support the engineering process (such as requirements, project plans, design modes, and test cases), while others are delivered alongside the thing being built (such as user manuals, shipping labels, and parts lists).

Shared assets

Shared assets are the "soft" artifacts associated with engineering lifecycle of the products, the building blocks of the products in the product line. In the early days of PLE this largely constituted software, but as PLE has matured the notion of shared assets has expanded to include a wide spectrum of engineering artifacts [16]. In fact, shared assets can be whatever artifacts are representable with software and either compose a product or support the engineering process to create a product. These can include, but are not limited to the following:

- 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 and parts lists

Means of production

The means of production is the mechanism turns shared assets into configured, product-specific instances that, together, constitute the artifact set for one of the products in the product line. Configuring the shared assets for each product in turn produces the entire set of products.

The means of production can be manual, but for product lines of any size or frequency of change, manual production is impractical; some form of automation is required.

Low-end automation might be a programming language's macro processor combined with compiler flags and #ifdef statements to turn blocks of code on or off. However, this scheme scales poorly, may not trace well across different kinds of artifacts, and in fact may not work for (for example) requirements or design models. This leads to an ad hoc collection of techniques for expressing variation, each specialized to its own type of shared asset.

At the high end of the spectrum are special-purpose PLE tools take a description of a product and automatically instantiate the shared assets to support the described product. Examples of such tools include Gears [14], pure::variants [2], XVCL [12], Dopler [6], and more.

It is at this end of the spectrum that our interest lies.

Characteristics of automation-centric PLE

McGregor [18] identifies five different kinds of automated approaches to product derivation, approaches that are differentiated by the way in which we tell the automation what product to build:

- Specification approaches. Here, a specification of the desired product is provided to the automation. The specification may be a description of the product's features, or the use cases that it satisfies, or some way of picking the product out of a domain of similar products.
- Intelligent build. Here, the automation is a build tool such as ant [19]. The product engineer simply directs the automation, via a script, to assemble products from available building blocks.
- Domain specific languages (DSLs) and product generation. In this approach, a programming language (one that includes abstractions with the expressiveness to capturing concepts in the products' domain) is used to write programs that are then fed to automation (purpose-built for the DSL, one presumes) that "implements" the solution.

Metamodeling, which is an approach used to create a DSL, and frame technology, a variant of metamodeling, round out the list.

Thus, a common concept of automation-based PLE is a description of the desired product that is fed to the automation.

Automation-centric PLE in practice

One particular manifestation of automation-centric PLE appears in the literature (and in practice) under the name "Second Generation Product Line Engineering (2GPLE) [16]. It came about largely because of the availability of commercial industrial-strength automation tools called *configurators* in the early 2000s. This is the model we will discuss in the following, which has quite a number of published case studies to recommend it [5][9][15][8][10].

Figure 1 illustrates the essence of 2GPLE with Gears [14] standing in as the configurator. This "factory's" supply chain is at the left, in the form of shared assets that are configurable because they include *variation points* – places where the artifact needs to be different depending on what feature set we are asking it to support – shown by the gear symbols in the figure. Variation points are expressed in terms of the *features* available in each of the products. A feature-based product specification at the top tells the configurator how to

configure the assets coming in from the left. The resulting products, assembled from the configured assets, emerge on the right. This enables the rapid production of any variant of any of the assets for any of the products in the portfolio. Once this production line capability is established, products are instantiated – derived from the shared assets by the configurator – rather than manually created.

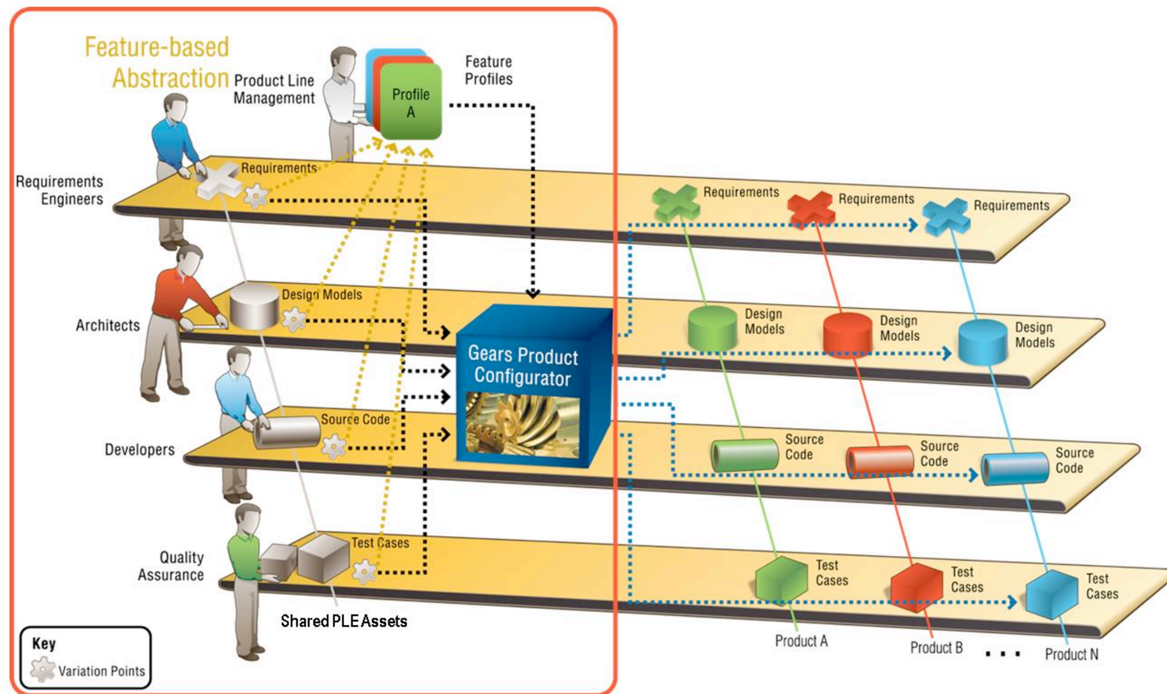

Figure 1. A PLE factory showing shared assets (left) being configured by a configurator (in this case, Gears), informed by feature profiles (top) that describe the product.

The 2GPLE approach has a number of important characteristics that distinguish it.

Characteristic 1: Features express product differences across all phases of the product lifecycle. A feature, to paraphrase [13], is a distinguishing characteristic of a product, usually visible to the customer or user of that product. An example is a function that one product can perform that others cannot. The concept of "feature" allows a consistent abstraction to be employed when making choices about a product, from conceptualization to deployment. Stakeholders throughout the entire product line environment are fluent in the language of features. Marketers sell features; testers test features; suppliers provide parts that realize features; software programmers implement features; requirements engineers specify features; and so forth. All of these roles are able to communicate meaningfully in this *lingua franca*, as opposed to the arcane languages of each one's discipline.

Feature declarations lay out the choices that are available to you. For example, when you buy a new car: Two door or four door? Sport package, luxury package, or economy package? Moon roof?

Feature assertions describe constraints and dependencies among the features. For example, feature assertions could express the constraint that a feature (or combination of features), if chosen, either requires or excludes the presence of another feature (or combination of features). Feature assertions are a critical aspect of building feature models; they capture the knowledge necessary to prevent unacceptable products from being constructed.

Feature profiles are used to reflect choices made from among the available features for the purpose of instantiating a product. A feature profile is associated with a subsystem or 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 feature assertions.

Characteristic 2: Shared assets from across the full lifecycle, with consistent variation management. Shared assets come from all lifecycle phases (not just the software, which was the focus in early PLE).

Shared assets in PLE are engineered to be shared across the product line. Shared assets are designed with built-in *variation points*, which are places in the shared asset that change depending on the product in which the shared asset is used. Variation points use variation mechanisms [1] to impart product line diversity; these mechanisms include macro's in code, substituting one variant of the artifact for another; runtime conditionals and configuration files; attributes and filters; model and text transformations; feature mappings, parameterization, and many more.

Further, however, 2GPLE enforces *consistent* treatment of all shared assets under the production infrastructure.

To see what this means, try to visualize the situation it without it. 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 Unified Modeling Language (UML) tool and has embraced inheritance as the mechanism for managing PLE design variations. The development team is using a FODA feature model drawn in a graphical editor, plus #ifdefs, build flags, and CM 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 artifacts from all across the lifecycle, but consistent and traceable treatment of variation, using the same language (based on feature choices) to express variation points in every kind of shared asset. Figure 2 shows the classic V-model for systems and software engineering. Each phase is augmented by the addition of variation points (indicated

by the gear symbol) to the artifacts native to that phase. A Bill-of-Features for a product corresponds to the feature selections within the feature profiles for that product.

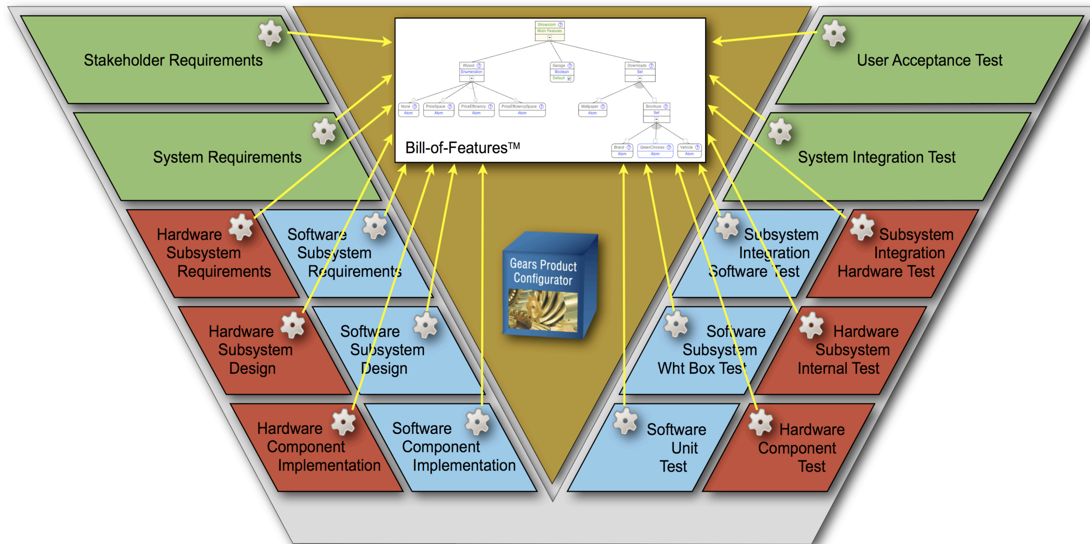

Figure 2: V-model for system engineering and PLE

Characteristic 3: CM that maintains shared assets, not products or asset instantiations. Under the 2GPLE discipline, the full superset of available PLE assets (and not the individual products or systems) are managed under configuration management (CM). A new version of a product is not derived from a previous version of the same product, but configured from the correct (possibly new) version shared assets themselves.

Characteristic 4: Support for product lines across organizational boundaries. Organizations, even small ones, often have separate divisions that work to support their product line. Large organizations almost certainly do. It is impractical to expect everyone to work on the same feature model, the same set of shared assets, and so forth. Certainly having one global collection of feature declarations for an entire production line is impractical when features may number in the hundreds or even thousands. It makes much more sense to modularize the feature model in a way that corresponds to the organizational structure of the enterprise. This lets engineers work largely independently within the confines of their own organizational units and domain expertise.

What must 2GPLE automation provide?

To power the factory of Figure 1, the first capability the configurator must obviously provide is the ability to exercise variation points in shared assets according to a specification of the product (which, in the 2GPLE paradigm, comes in the form of feature profiles).

Clearly there must also be tooling that supports the construction and management of feature models (including feature declarations, assertions, and profiles).

In the 2GPLE realm, the tooling must assist in the construction of variation points – for instance, to provide a structured editor to help PLE engineers write feature expressions, as opposed to forcing them to write the expressions by hand. All of this pre-supposes a structured, learnable variation point language and a relatively simple menu of variation point mechanisms to invoke.

It also needs to enforce the assertions it has captured – that is, report when a product is specified that violates one or more feature constraints.

Finally, it must have a user interface that scales for work across organizations with thousands of engineers whose experience working with engineering tools (and many different tools among them) will likely vary greatly.

Satisfying this list might be a tall order but configurators such as Gears combine all of these capabilities into a single tool today.

If that is the case, then where is the PLE ecosystem on which this paper is predicated? The next two sections add requirements on the tooling that even a hypothetical PLE supertool could not meet.

Letting users continue to use the engineering tools they're used to

In a PLE context, requirements engineers work on requirements, software engineers work on software, test engineers build test cases, assembly engineers build bills of materials and parts lists, tech writers write user manuals, technical managers build engineering plans, build engineers craft build scripts, and so forth. They do these things in the context of the entire product line and not just individual products, but by and large their jobs remain the same.

Good luck to any PLE tool purveyor who comes into an organization and announces that, in order to use the PLE tool being proffered, everyone must drop the tools they are using to do their daily work and use the PLE tool instead.

And yet, we want the requirements engineer, software engineer, test engineer, and the rest to put variation points into their artifacts, and we want that process to be assisted and facilitated by automation that will eventually exercise those variation points.

This means we need way to support the specification and selection of variation in assets and artifacts from across the entire spectrum of the product life cycle – in, for example, DOORS requirements modules, Word documents and Excel spreadsheets, build files for Make or Ant, Rhapsody UML models, and many more.

But off-the-shelf lifecycle management tools do not know about PLE or variation points or feature models or have interfaces that allow for variation points to be exercised based on feature choices made in another tool. And no PLE tool vendor is going to replicate the interface and functionality of, for example, IBM Rational's DOORS requirements tool in their PLE tool (to name but one example of a requirements engineering tool in but one life cycle area), let alone all the other possibilities.

The solution, of course, is for the PLE tool to interface with the life cycle tools to support variation. There are fundamentally three ways to achieve variation in an asset, depending on what you know about the digital representation of the associated artifact:

- The representation of the artifact is proprietary and closed, or editors for it are not available or are impractical. For example, if our products include a picture that is

different from product to product, some of our artifacts may be JPG files. Our only avenue to achieve variation in this case is to have the variation point can simply choose from a selection of alternative variants, using each one as is, as opposed to trying to change the picture by editing the image, pixel by pixel, stored in a one-size-fits-all superset picture file.

- The representation of the artifact is "open," so that you can change it using an available open market tool. For example, artifacts stored as simple text files may be transformed (during the exercising of their variation points) by simple word or line substitution. Artifacts that are Microsoft Word documents stored in Office Open XML format can be transformed by third party tools.
- The representation of the artifact is proprietary, but the owning organization offers a business relationship to allow your PLE tool to edit their artifacts. Suppose your requirements are stored in DOORS, using hundreds and hundreds of DOORS requirements objects. The representation of those objects is proprietary, but using the strategy in the first bullet is out of the question: Swapping in and out whole requirements documents that each differs by just a little bit is untenable. It would be much better to make an arrangement with the vendor (IBM Rational in this case) so that you can write a piece of software that can insert and operate on variation points throughout the DOORS representation of a body of requirements.

This adds another requirement to the product line engineering automation engine that does not come immediately to mind: does its proprietor have the necessary business relationships to support variation in the third category? This property is essential for the property of 2GPLE we described earlier: support for variation in shared assets originating from across the entire life cycle.

And now we have an ecosystem: The PLE automation must partner with (interface with) the vendors (and their tools) that provide the functionality that our PLE engineers use in their day-to-day jobs.

Ecosystems

An ecosystem is "a system, or a group of interconnected elements, formed by the interaction of a community of organisms with their environment" or, perhaps more helpfully to our context, "any system or network of interconnecting and interacting parts, as in a business" [7].

Much has been written about software ecosystems [3], which are ecosystems in which the elements are software or software-related entities, such as the organizations involved in the production of software.

McGregor and Monteith point out that ecosystems are believed to contain the necessary elements to sustain life of the ecosystem's elements [17]. If the elements have value to us, then the ecosystem also has value to us, and is worth studying to understand how it sustains that life and (where we have control over it) what we can do to enhance that sustainment.

In this paper, we use the term ecosystem in reference to tools, technologies, products, and suppliers of all of those that, together, provide an industrial-strength PLE technology solution.

There's still one final piece to add to our ecosystem picture: It has to be able to handle the three dimensions of PLE.

Ecosystem support for three dimensions of PLE

Organizations building a product line have to deal with three concerns, which are illustrated in Figure 3. They have to:

1. Manage the plurality of products (outward-pointing axis). The 2GPLE paradigm discussed earlier, and the automation-centric PLE approach for which 2GPLE is a specific instance, is aimed largely at this dimension. This is the realm of mainstream PLE tools such as the ones mentioned previously.

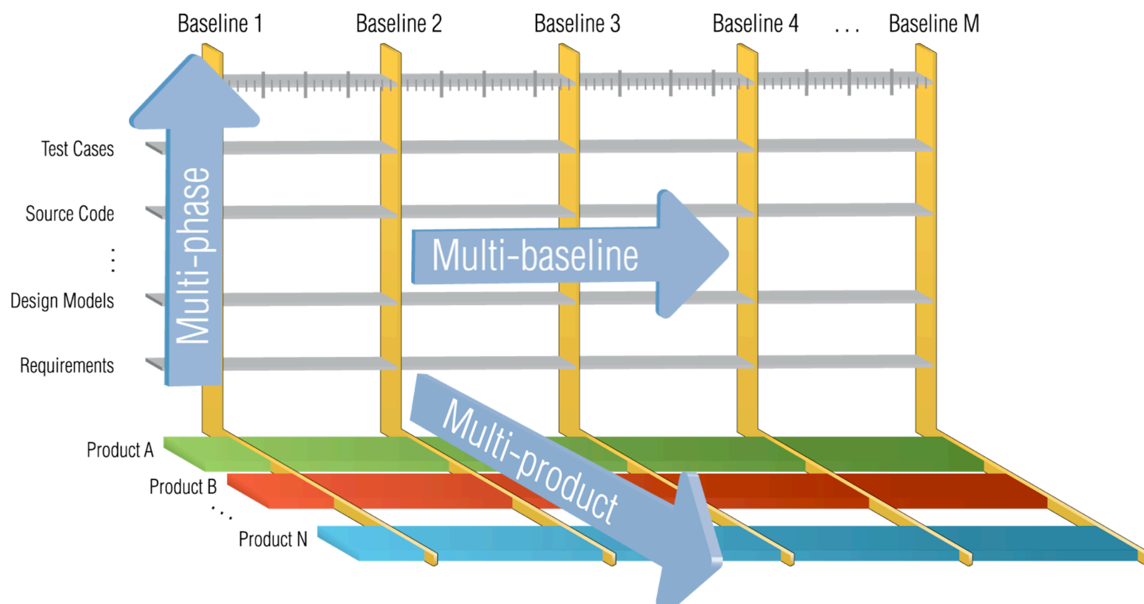

Figure 3 Three dimensions of product line engineering.

2. Evolve the portfolio over time (horizontal axis). The *Multi-baseline* axis of Figure 3 deals with the usual temporal concerns of product engineering, such as version, configuration and change management. Configuration management (CM) systems are brought to bear to track the evolution of shared assets over time. Just as no PLE tool is going to subsume all requirements engineering or testing tools, no PLE tool is going subsume all CM system capabilities. At best, they will be compatible with general CM systems or even be agnostic to the CM systems in use by simply working on files checked out into workspaces. But to handle the critical need to manage evolution, we now add CM systems to our PLE ecosystem.
3. Manage the shared assets and the products to which they contribute across lifecycle phases or disciplines (vertical axis). The ecosystem we discussed in the previous section is aimed largely at this dimension. However, in a true systems engineering environment, traceability across the lifecycle artifacts (e.g., from requirements to code to tests, or from design models to parts list) plays a critical role. That traceability must be maintained as the product line and its shared assets evolve over time, and must persist when shared assets' variation points are exercised to derive products.

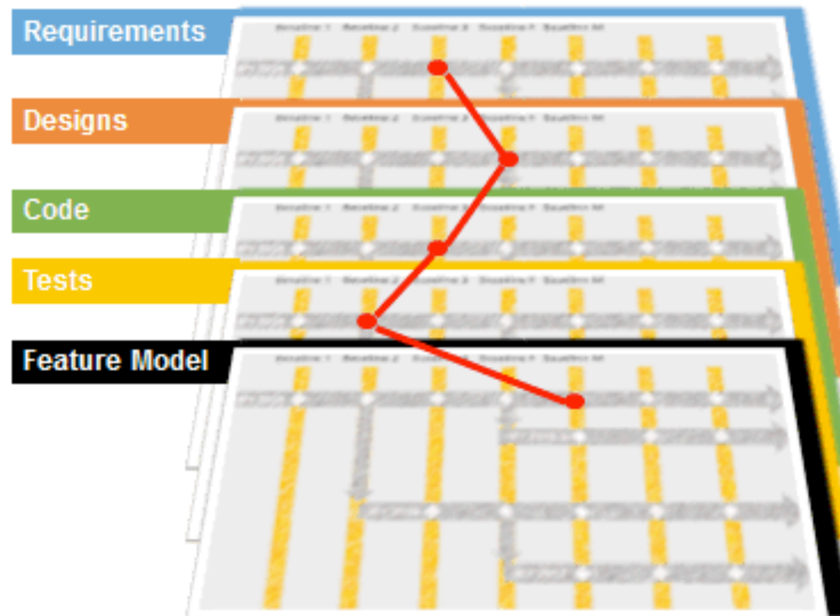

Figure 4. Federated global configuration management

Figure 4 illustrates the last point. It shows four kinds of shared assets plus the product line's feature model, each possibly stored in separate physical repositories (e.g., a DOORS database, a code library, etc.), all evolving over time (left to right). The red "zigzag" line connects versions of shared assets and the feature model that correspond to a consistent build set from which products are derived at a point in time. The zigzag line is called a "temporal context." To build a product, we check out the right versions (identified by the zigzag) of the shared assets and feature model, and use our configurator to derive the desired product. The result is a set of asset instances, as usual, with their trace links still in place, to produce a consistently linked product.

The final piece of technology to add to our PLE ecosystem is the technology to manage this traceability over time and across engineering artifacts. This technology need is being addressed by technology vendors. IBM's Global Configuration Management solution is a leading example:

> *"A global configuration assembles other configurations within and across applications. The other configurations come from contributing applications, such as Rational Team Concert™, Rational® Quality Manager, Rational Design Manager, Rational DOORS® Next Generation, or other global configurations.*
>
> *"Global configurations assemble all the relevant artifacts (requirements, design models, test plans, source code, other global configurations) for a specific component release or product version. Use the assembled version to work in parallel on multiple development streams, re-create a development environment from the past, or build a hierarchical component structure of a system that you are developing, while managing versions and variants across product lines." [11]*

Summary

This paper has tried to show that a holistic solution to product line engineering in the large involves, by necessity, an ecosystem of collaborating technologies. The three dimensions of PLE – multi-product, multi-phase, and multi-baseline – each bring players and their technologies into the PLE ecosystem. The multi-product dimension is the domain of the PLE tools that may first to mind when designing an automated PLE solution. The multi-phase dimension requires us to consider life cycle tools such as modeling or testing tools, as well as consider cross-life-cycle traceability. And the multi-baseline dimension adds change and configuration management technologies to the ecosystem.

PLE is the one engineering realm that brings together all of these communities.

For those in the market for a PLE technical solution, they must keep in mind the multi-product, multi-phase, multi-baseline ecosystems whose inter-operation is the key to required functionality over time and across their product domain.

On the other side of the coin, technology providers who want to succeed in the PLE arena must also take into account the PLE ecosystems in which they are going to serve, and make their technologies open enough to be compatible with the other automation with which they must cooperate, and be open to business relationships with other technology providers in the ecosystem. It is a daunting challenge, but one that is being met by the engineering community.

References

[1] Bachmann, F., Clements, P. “Variability in Software Product Lines,” Technical report CMU/SEI-2005-TR-01, Software Engineering Institute, 2005.

[2] Beuche, D. Modeling and building software product lines with pure: Variants. In Proceedings of the 15th International Software Product Line Conference, Limerick, Ireland, Sept 08–12, 2008, ACM Press, 2008; SPLC ’08, 358.

[3] Boucharas, V., Jansen, S, Brinkkemperm S, Formalizing Software Ecosystem Modeling. In Proceedings of the 1st international workshop on Open component ecosystems (IWOCE '09), 2009, pp 41-50.

[4] Chastek. Gary, Donohoe. Patrick, and McGregor. John, "A Study of Product Production in Software Product Lines," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Note CMU/SEI-2004-TN-012, 2004. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6901

[5] Clements, P., Gregg, S., Kreuger, 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,* Jan-Feb 2014.

[6] Dhungana, D.; Rabiser, R.; Grunbacher, P.; Lehner, K.; Federspiel, C. DOPLER: an adaptable tool suite for product line engineering. In Proceedings of the 11th International Software Product Line Conference, Kyoto, Japan, Sep 10–14, 2007; SPLC ‘07, Second Volume 151–152.

[7] Dictionary.com, accessed November 10, 2015.

[8] 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.

[9] 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.

[10] 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.

[11] IBM, "Global Configuration Management," https://jazz.net/help-dev/clm/index.jsp?re=1&topic=/com.ibm.rational.gcapp.doc/topics/c_gcm_node_managing_global_configs.html

[12] Jarzabek, S.; Zhang, H. XML-based method and tool for handling variant requirements in domain models. In Proceedings of the 5th International Symposium on Requirements Engineering, Aug 27–31, 2001; Toronto, Canada, 2001; RE'01, 166–173.

[13] 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.

[14] Krueger, C. The Systems and Software Product Line Lifecycle Framework. BigLever Software Technical Report #200805071r3. 2010. http://www.biglever.com/extras/Spl-LifecycleFramework.pdf

[15] Krueger, C., Churchett, D., Burhdorf, R. "HomeAway's Transition to Software Product Line Practice: Engineering and Business Results in 60 Days," 2008 Software Product Line Conference, San Francisco.

[16] Krueger, C., Clements, P. "Systems and Software Product Line Engineering," *Encyclopedia of Software Engineering*, Philip A. LaPlante ed., Taylor and Francis, 2013.

[17] McGregor, J., Monteith, Y. "A Three Viewpoint Model of Software Ecosystems," *Proc.* IASTED *Software Engineering Applications Conference*, 2012.

[18] McGregor. John, "Preparing for Automated Derivation of Products in a Software Product Line," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, , Technical Report CMU/SEI-2005-TR-017, 2005.

[19] The Apache Software Foundation. Axis CPP - Ant Build Guide. http://ws.apache.org/axis/cpp/antbuild-guide.html (2004).

Biography

- **William J. Bolander** earned his bachelor's degree in mechanical/electrical engineering from General Motors Institute, now Kettering University and his master's degree in mechanical engineering from Purdue University. After graduation in 1984, he joined GM developing engine control algorithms and calibration. In 1996 he became the Manager of GM Powertrain's Algorithm Engineering department. In 2000,

Bill was promoted to GM Technical Fellow responsible for GM Powertrains Global Control Engineering Processes. In this role, he led the Controls Engineering Process Group responsible for process improvement activities, including ISO 9001 and CMMi. In 2009, Bill's role expanded to include all of GM's Electrical, Controls and Software development, for both Powertrain and Vehicle Product Development, where Bill lead a program to update and converge GM's engineering processes and tools used globally. In 2013, Bill joined the IBM Rational team as a Global Automotive Solution Executive where he is helping the industry adopt smarter product development processes and tools. Bill holds 16 US patents for automotive related innovations. These contributions to GM's technology have earned him four "Boss" Kettering Awards, GM's highest recognition for engineering invention. In 1995, Bill was the first winner of the $500,000, Lemelson-MIT Prize, the nation's largest single prize for invention and innovation.

- **Dr. Paul C. 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. He received his PhD in computer sciences from the University of Texas at Austin.

- **Dr. Charles Krueger** is an acknowledged thought leader in the PLE field, with 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 proven expertise in leading commercial product line development teams, and 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.