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

# Model Based Engineering and Product Line Engineering: Combining Two Powerful Approaches at Raytheon

Dr. Bobbi Young
Raytheon Company
bobbi.young@raytheon.com

Dr. Paul Clements
BigLever Software
clements@biglever.com

**Abstract.** Model based engineering (MBE) refers to a systems engineering approach that employs models as an integral part of a system's engineering stream, providing a formality and semantic rigor that lends itself to analysis and prediction, thus enabling earlier detection of problems. Product line engineering (PLE) is a way to engineer a portfolio of related products in an efficient manner, taking full advantage of the products' similarities while respecting and managing their differences. Raytheon, one of the world's largest defense contractors, is applying PLE and MBE together and combining the benefits of each. This paper will show how Raytheon is using the two methodologies to support each other, and the lessons it is learning as it does so.

## Introduction

Model based engineering (MBE) refers to a systems engineering approach that employs *models* as an integral part of a system's engineering stream. Models typically bridge the gap between engineering activities, and provide a formality and semantic rigor that lends itself to analysis and prediction, thus enabling earlier detection of problems. The formality can also lend itself to automation that can transform a model into another formal representation of the system, typically, one that is "closer" to the final implementation or realization. This automation can reduce the time, effort, and errors that are associated with the manual translation that would otherwise be required.

Systems and software product line engineering, or "product line engineering (PLE)" for short, is a way to engineer a portfolio of related products in an efficient manner, taking full advantage of the products' similarities while respecting and managing their differences. Considering a portfolio as a single entity to be managed, as opposed to a multitude of separate cloned products to be managed, brings enormous efficiencies in production and maintenance; these efficiencies are delivering order-of-magnitude improvements in engineering cost, time to market, staff productivity, product line scalability, and quality [10].

What happens when an organization tries to apply both of these groundbreaking, organization- changing methodologies at the same time? Can they work together at all? This paper conveys the experience of Raytheon, one of the world's largest defense contractors, as it is seeking to apply PLE and MBE together.

## What Is Product Line Engineering?

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

Born in the 1980s in the software field, but now having grown well beyond those early roots, PLE offers large savings observed from engineering the whole family rather than separately engineering each member. Numerous case studies [1][5][6][11][15][16] show that exploiting the commonality throughout the products’ total life cycles can return substantial improvements in time to market, cost, portfolio scalability, engineer productivity, and product quality [14]; no other engineering paradigm shift has, to our knowledge, brought about results that rival these.

Modern PLE is based on the concept of *features* that describe products and relies on automated configuration tools (*configurators*) to instantiate *shared assets* (engineering artifacts) in order to support specific products.

## *PLE as a factory*

An analogy with factory-based manufacturing serves to illuminate the important concepts. Manufacturers have long used engineering techniques to create a product line of similar products using a common factory that assembles and configures parts to produce the varying products in the product line. For example, automotive manufacturers can create thousands of unique variations of one car model using a single pool of parts carefully designed to be configurable and factories specifically designed to configure and assemble those parts.

In PLE, the configurator is the factory’s automation component; the “parts” are the assets in the factory’s supply chain. A statement of the properties desired in the end product tells the configurator how to configure the assets.

Figure 1 illustrates. This factory’s supply chain is at the left, in the form of shared assets that are configurable because they include variation points that are expressed in terms of the features **[9]** available in each of the products. A 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 – rather than manually created.

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 requirements, engineering models, or development plans) associated with the engineering process that produces them.

In this context “product” means 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). These artifacts are the product line’s assets.

Assets can be whatever artifacts are representable digitally and either constitute part of a product or support the engineering process to create a product. Four kinds of shared assets are shown in Figure 1, but those are just examples. Shared assets can include, but are not limited to, requirements, design specifications, design models, source code, build files, test plans and test cases, user documentation, repair manuals and installation guides, project budgets, schedules, and work plans, product calibration and configuration files, data models, parts lists, and more. Assets in PLE are engineered to be shared across the product line.

In this paper we will focus on models and designs as the shared assets of paramount interest, as those lie at the intersection of MBE with PLE.

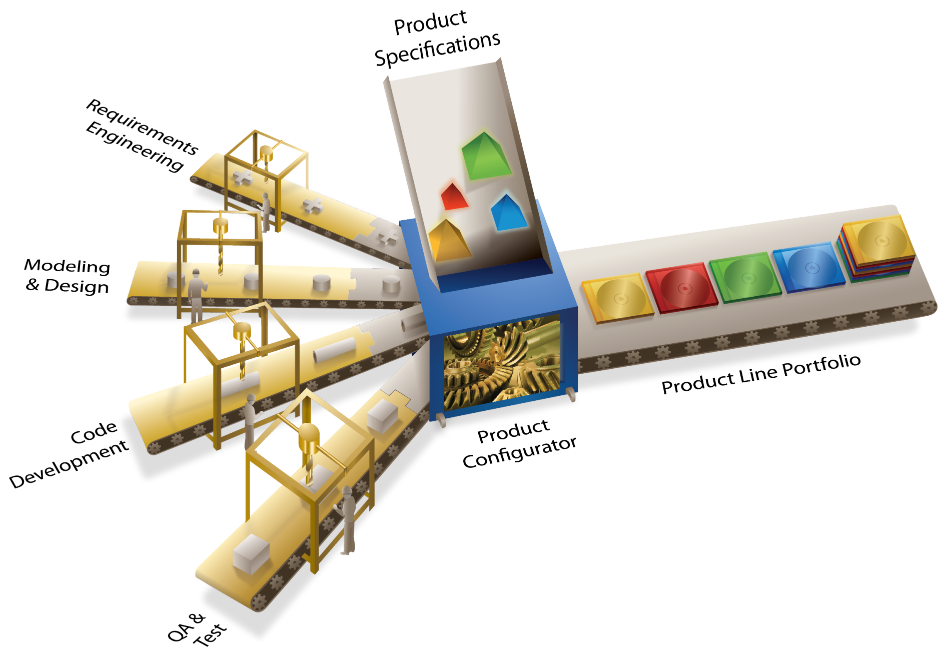


Figure 1: PLE as a factory.

## *PLE contrasted with product-centric development*

PLE stands in contrast to traditional 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 after the initial clone operation. In particular, it derives very little benefit from commonality in a product's sustainment or maintenance phase, where data show most products consume up to 90% of their project resources.

Figure 2 shows a production shop in which N products are developed and maintained. In this stylized view, each product comprises requirements, design models, source code, and test cases. Each engineer in this shop works primarily on a single product. When a new product is launched, its project copies the most similar assets it can find, and starts adapting them to meet the new product's needs.

Coordination among projects, if there is any at all, is ad hoc and de-centralized, meaning that, each of the N product teams should really confer with each of the other N-1 product teams. These communication paths are shown in red in Figure 2. This communication obligation imposes an overhead that grows as the square of the number of products. This complexity will quickly overwhelm any engineering staff; in order to get their products out the door on time and on budget, each product team will focus more on their product silo and less on taking advantage of the commonalities and interdependencies among the other products. The result is divergent product silos, low degrees of sharing, and high duplication of effort across the product silos to fix the same defect multiple times in multiple products, or to independently implement the same enhancements in different ways in different products.

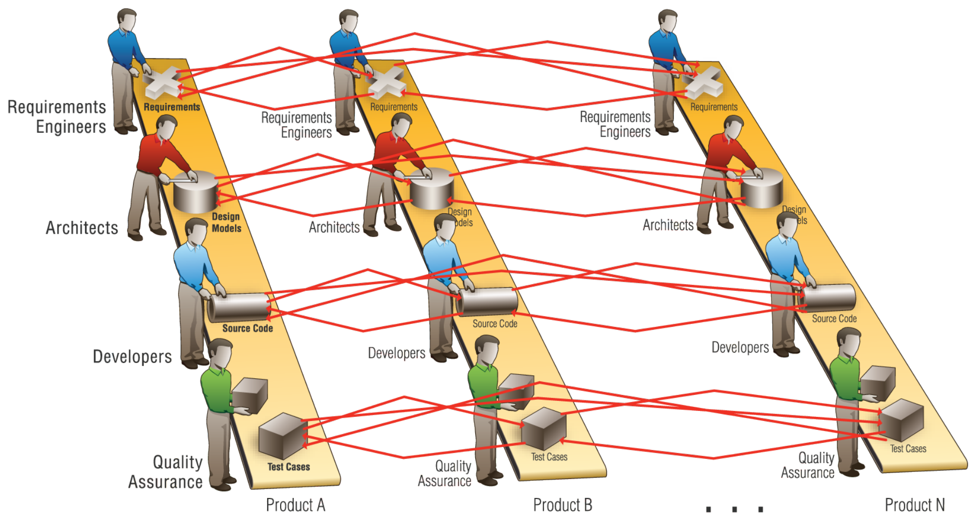


Figure 2: Product-centric development yields O ($N^2$) complexity

Under the PLE approach, all development takes place in the factory and not in project silos. This assures the maximum amount of cross-project sharing on an ongoing basis. Coordination happens between products and the factory, which for a portfolio of *N* products is an O (*N*), as opposed to an O ($N^2$), proposition.

## Model Based Engineering

Model based engineering is "an approach to engineering that uses models as an integral part of the technical baseline that includes the requirements, analysis, design, implementation, and verification of a capability, system, and/or product throughout the acquisition life cycle" [Final Report, Model-Based Engineering Subcommittee, NDIA, Feb. 2011].

A model, in turn, is "a physical, mathematical, or otherwise logical representation of a system, unity, phenomenon, or process" [DoD 5000.59-M 1998].

Model based engineering is held in contrast to approaches in which documents, typically in prose, serve as the basis for the information exchange among stakeholders in a systems engineering process. An example is the hand-off of a design document from designers to implementers or manufacturers.

Models, because of their physical or mathematical formulation, should be less ambiguous by nature, and therefore more amenable to high-confidence analysis, thus reducing errors and re-work in the systems engineering process. Some models are carry a precise enough semantics to enable automation to translate them into engineering artifacts downstream in the engineering flow – code generation, for example, in the case of models that represent designs for software, or manufacturing prototypes in the case of models that represent physical entities. The formality of the models (coupled with confidence in the automated translation) allows us to have confidence that the downstream information is consistent with or carries out the directives of the upstream model, and is therefore valid and acceptable.

## MBE and PLE at Raytheon

### *Our Example: Integrated Air and Missile Defense (IAMD)*

MBE and PLE are both in use separately, to varying degrees, throughout Raytheon. To maintain confidentiality we will illustrate how the two paradigms can work together by focusing on a particular (fictitious but realistic) example from the IAMD domain: a system for integrated air and missile defense called GloboShield. GloboShield's mission is to provide a fully integrated capability to protect a theater from air and missile attack by detecting, tracking, identifying, and destroying airborne threats. Such a system includes sensors, displays, planning functions, threat evaluation, health and status monitoring, communication with other friendly command and control systems for information exchange, and more.

Figure 3 is a sketch of a system architecture for GloboShield, identifying its major subsystems along with some explication about each. (Figure 3 does not tell the whole architectural story, of course. It does not show, for example, behaviors of or execution-time relationships among the subsystems; other architectural views do that. For the purposes of our discussion, we will let Figure 3 stand in for the entire range of useful architecture documentation.)

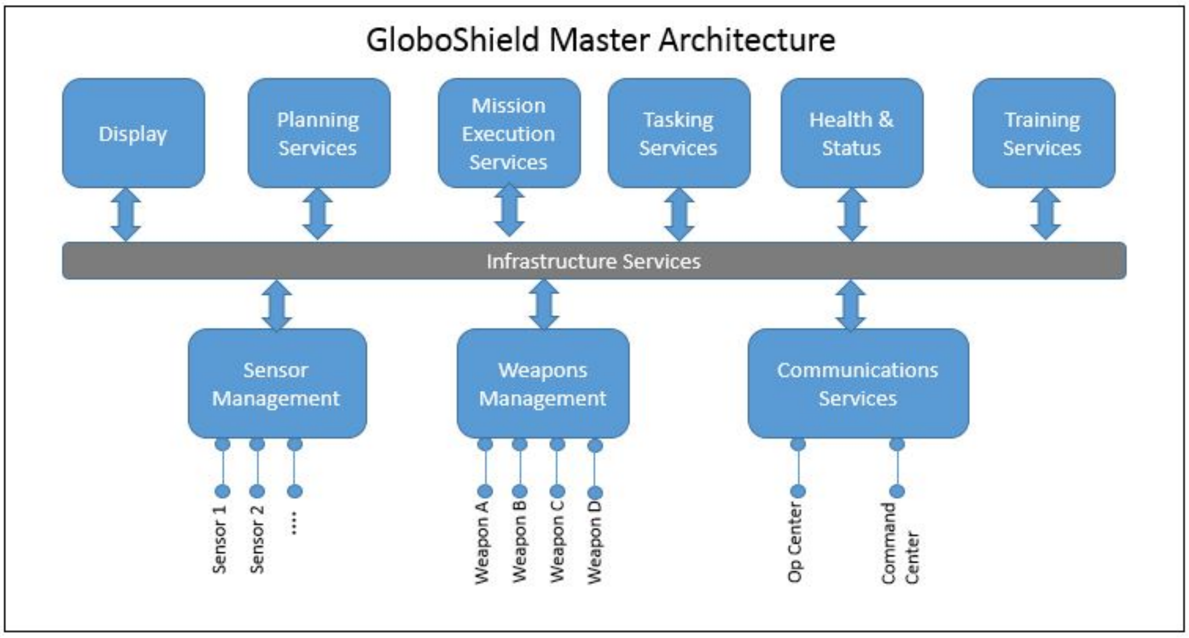


Figure 3: GloboShield Master Architecture Identifying Architectural Components

GloboShield is not a single product, but a product line. Customers can order GloboShield in different configurations, and with different levels of capability. For example, each product has different sensor and weapon systems, as well as different organizations for managing air and missile defense.

In addition, GloboShield provides options for its Threat Assessment capability. The customer may

- Choose or omit the Threat Determination service to identify a threat that could be an air-breathing target (ABT) and/or a theater ballistic missile (TBM).

- Choose or omit, in addition to the Threat Determination service, a Threat Ranking and/or a Threat Warning service.

Each instance of GloboShield will have its own system architecture that reflects the product choices outlined above as well as many others, but in all cases one that is derived from the architecture illustrated in Figure 3. Figure 4 shows four product architectures – that is, architectures for four product instances of GloboShield.

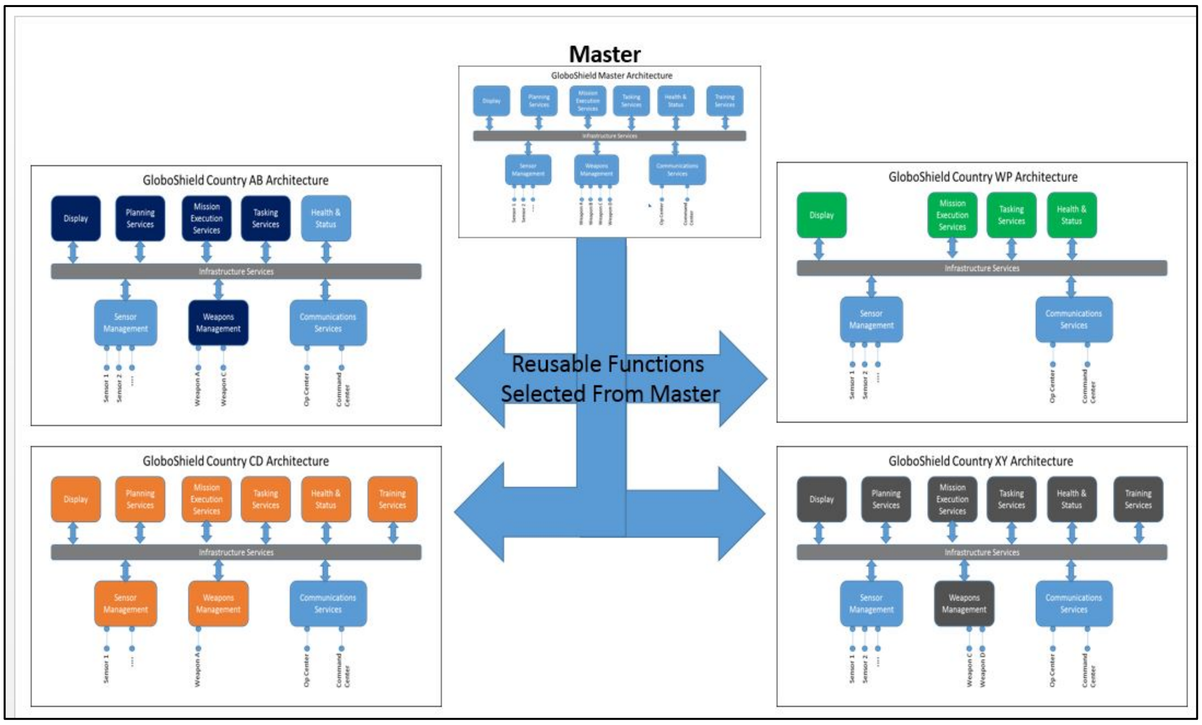

Master
GloboShield Master Architecture
GloboShield Country AB Architecture
GloboShield Country WP Architecture
GloboShield Country CD Architecture
GloboShield Country XY Architecture
Display
Planning Services
Mission Execution Services
Tasking Services
Health & Status
Training Services
Infrastructure Services
Sensor Management
Weapons Management
Communications Services
Reusable Functions Selected From Master
Sensor 1
Sensor 2
Weapon A
Weapon C
Weapon D
Op Center
Command Center

Figure 4: GloboShield Product Architectures Derived from the GloboShield Master Architecture

For the purposes of understanding how PLE and MBE can work together, it is not important to understand the details of the "master" architecture nor the details of the derived architectures in Figures 3 and 4. It is only important to understand that the product architectures represent derivations of the master architecture, typically in a superset/subset relationship. For example, a product architecture may omit some of the components that populate the master architecture (and therefore the relationships or "connectors" that tie those components to other components). This typically occurs when those components provide a capability that has not been chosen for a product. A product architecture may vary from the master in other ways; for example, a component may exist in both the master and a derived product architecture, but that component will be of a different "flavor" in the product, or bind certain choices about it that are available in its master-architecture analog.

## *MBE to Manage the Architecture*

Figure 4 is a *documentation*-based representation of the difference between a master architecture and product architectures that derive from it. In this world, viewgraphs are often the communication medium – an old quip holds that the world's most popular architecture description language is PowerPoint – in which the notation is often informal at best.

To move from the documentation-centric realm into the model-based realm, these architectural designs need to be captured in a more formal representation. Raytheon, like many systems engineering organizations, uses the System Modeling Language (SysML) [http://www.omg.org/spec/SysML/1.4/, retrieved October 2016] as its preferred language in which to represent system architectures. The architectures in Figure 3 and 4 can be represented in SysML. Raytheon uses Rhapsody from IBM Rational as the modeling tool with which to capture SysML models. In alignment with MBE, these choices enable analysis and derivation (for example, code generation) to be brought to bear, whereas any utilization of the architectural information before was purely manual and fraught with error-prone and labor-intensive work.

If this paper were only about MBE, we could end our story here, celebrating the benefits of MBE for each of our products. There would be (in our case) not one but five models (the master and the four derivatives). If models are good, then five models surely represent an embarrassment of riches.

## *PLE to Manage the Variation*

MBE, of course, is only the beginning of the story. GloboShield is not a plethora of separate products. It is a product line, a single family of similar systems with variation among them. At the very least, we want to reuse the parts that are common across the instances.

Reuse may come in the form of reusing systems, hardware, and/or software components which includes all their assets (requirements, designs, test, etc.). Under a reuse paradigm, usually each program develops an architecture and determines what they could reuse from other programs. Many times, the architecture is captured using block diagrams in slideware. Several programs in a family of similar programs will have very similar block diagrams but have different names for the same subsystem or component. Trying to identify potential reuse between a family of products becomes difficult and confusing, and we end up with the O ($N^2$) quagmire of Figure 2.

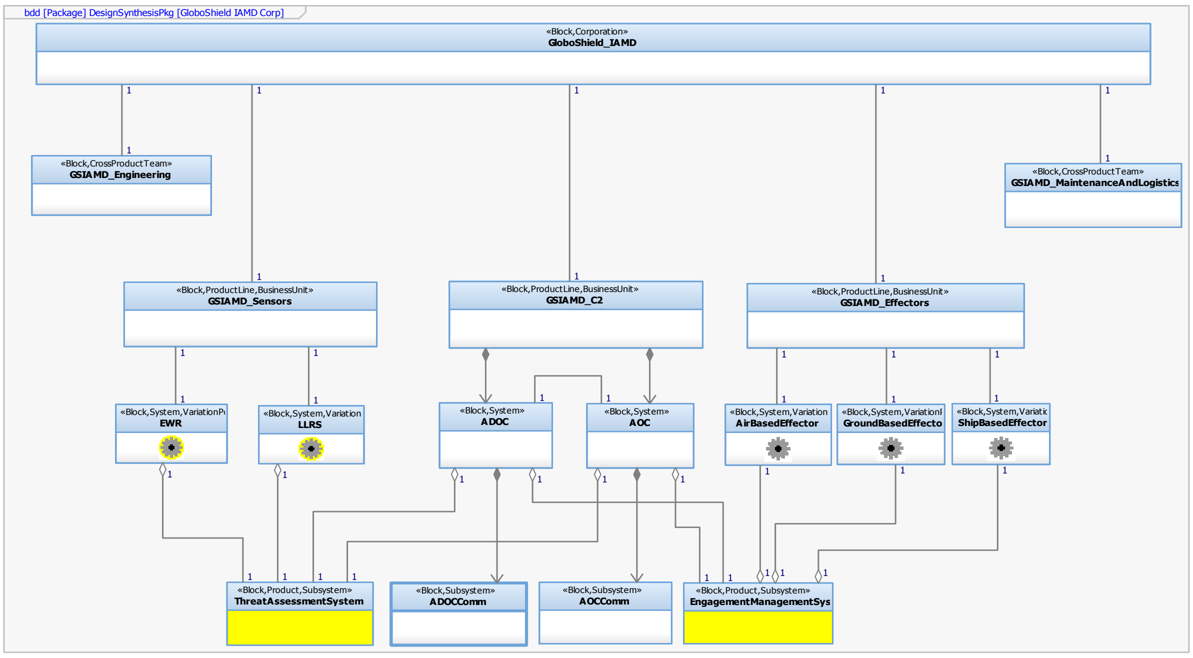
Indeed, when the systems engineering team first started to model the system architecture, variation was managed using typical methods such as stereotyping, inheritance, plug and play software components, and reusable interfaces. This approach worked up to a point for structural views of the architecture but could not easily handle the behavioral views. Each combination of variations could potentially produce different behavior scenarios. The only way to manage the scenario variations were to create separate views for each behavioral thread which became difficult to manage in the same model. There was no mechanism to automatically filter out the parts of the behavior that did not apply to a particular product. This led to creating different system models for each product that, in turn, had to be maintained like a clone and own solution.

Therefore, instead of aiming for simple reuse from program to program, we want to treat our architecture models in accordance with the PLE Factory model of Figure 1. There needs to be an approach to capture and manage the variations present among the products.

Using the factory approach, the systems engineering team developed a superset view of the architecture to capture those systems that would be variant. Figure 5 captures the variations identified for the structure of the enterprise view. The gear icon on some of the blocks along with the Variation Point stereotype denotes model elements that are variant – that is, that do not appear in every single member of the product line.

In order to identify the variant features, a feature model was developed. Raytheon has chosen the Gears PLE tool and framework from BigLever to serve as its feature modeling tool and the PLE Configurator shown in Figure 1 to power the PLE Factory.

The feature model captures distinguishing characteristics that set products apart – that is, only those features that represent variation are modeled. The feature model is a hierarchical decision tree that identifies and defines opportunities for variation; making selections in such a decision tree defines a particular product.



bdd [Package] DesignSynthesisPkg [GloboShield IAMD Corp]
«Block,Corporation» GloboShield_IAMD
«Block,CrossProductTeam» GSIAMD_Engineering
«Block,CrossProductTeam» GSIAMD_MaintenanceAndLogistics
«Block,ProductLine,BusinessUnit» GSIAMD_Sensors
«Block,ProductLine,BusinessUnit» GSIAMD_C2
«Block,ProductLine,BusinessUnit» GSIAMD_Effectors
«Block,System,VariationP EWR
«Block,System,Variation LLRS
«Block,System» ADOC
«Block,System» AOC
«Block,System,Variation AirBasedEffector
«Block,System,VariationI GroundBasedEffecto
«Block,System,Variatic ShipBasedEffector
«Block,Product,Subsystem» ThreatAssessmentSystem
«Block,Subsystem» ADOCComm
«Block,Subsystem» AOCComm
«Block,Product,Subsystem» EngagementManagementSys
1

Figure 5: GloboShield Block Definition Diagram Superset with Variation

Figure 6 shows part of the GloboShield feature model from Gears. The Gears development environment provides the ability to create feature models, define feature assertions (dependencies/constraints between features), feature profiles (a set of choices made against a feature model), identification of shared assets (such as an architectural model), and defined configuration of products. The feature model is a hierarchical tree structure that represents product portfolio options. It describes the variant features a customer may choose from for a specific product. Features defined in the feature model can be capabilities (i.e., functional, operational, presentation, implementation techniques, operating systems, and operating platforms) that represent variation among products. Each leaf node of the tree represents a customer feature option. Some options may be dependent on other options and need to be included or cannot be combined with other options. These dependencies and constraints are defined as assertions and are captured as rules.

Feature profiles are defined from the feature model by selecting the feature options that define a specific product. Figure 7 shows a portion of a feature profile that chooses the Threat Determination option for a GloboShield product instance.

Figure 8 shows a different feature profile that selects Threat Determination, Threat Warning, and Threat Ranking for a different GloboShield product instance.

Feature choices must be mapped to the variation within shared assets, so that the configurator can reflect the feature choice in the configuration of the shared asset. This is accomplished by adding variation points to the assets.

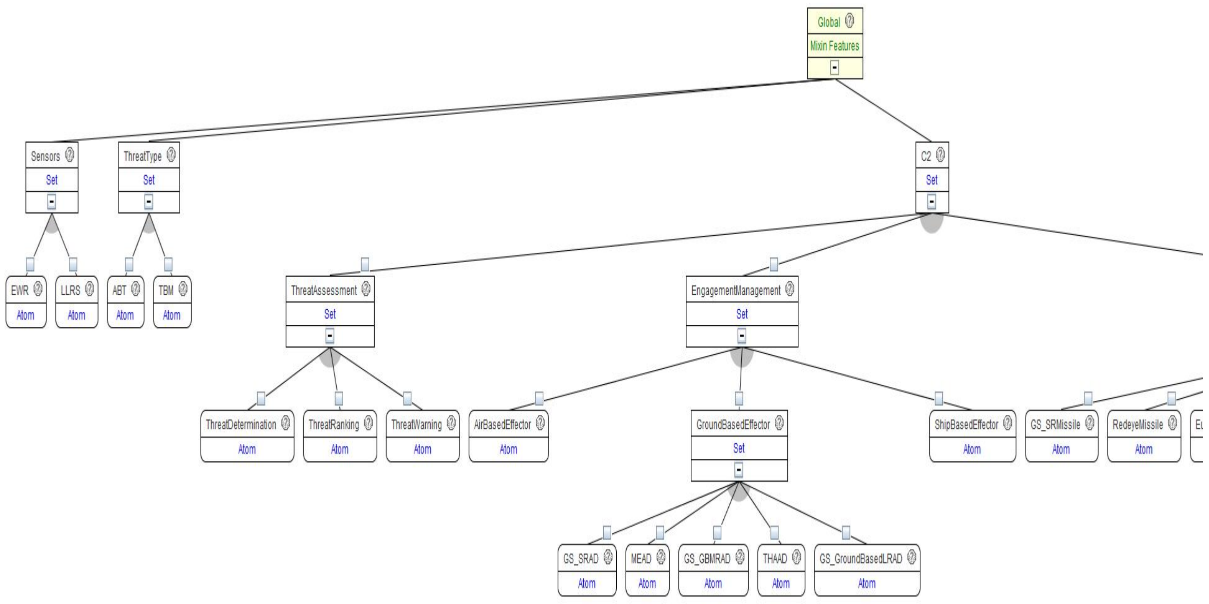


Figure 6: Feature Model for Threat Assessment (excerpt)

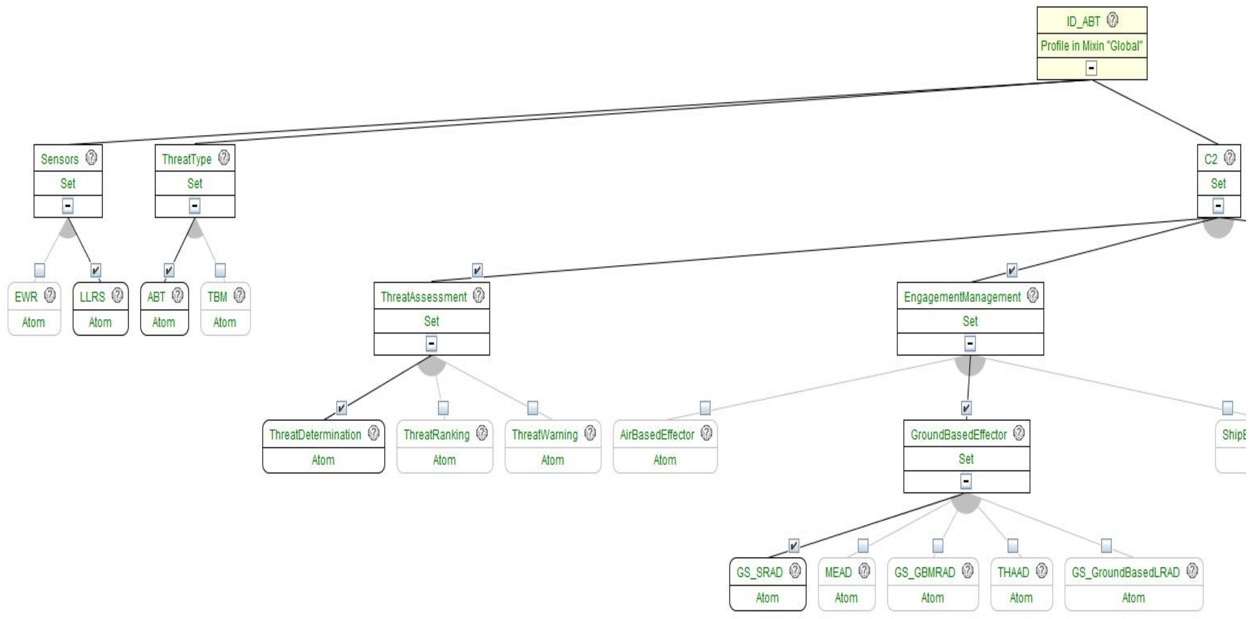


Figure 7: Feature Profile For Threat Determination (excerpt)

Figure 9 shows a variation point that has been added to the ThreatDetermination_COTS block in Rhapsody. The variation point encapsulates that model element that is identified as a variation point along with variation point logic. The variation point logic is written in a special-purpose Gears logic language that maps to the feature declarations. The logic tells Gears what feature choices or combination of feature choices will cause that block to be included in the projection, or product-specific instantiation of this shared asset.

The Gears configurator takes a feature profile name as input and configures the shared assets accordingly by exercising their variation points according to the feature choices in the profile. The configurator instantiates the projection for each variation point the feature parameters values in the profile. Actuating all the variation points in each shared asset configures the complete instance of a product.

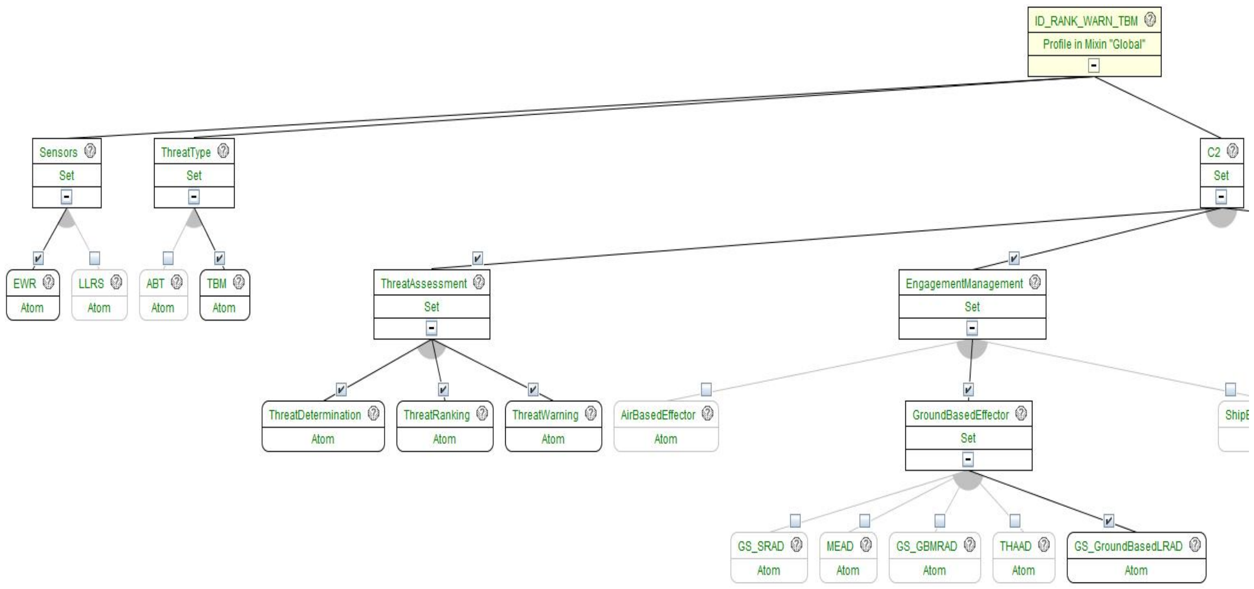
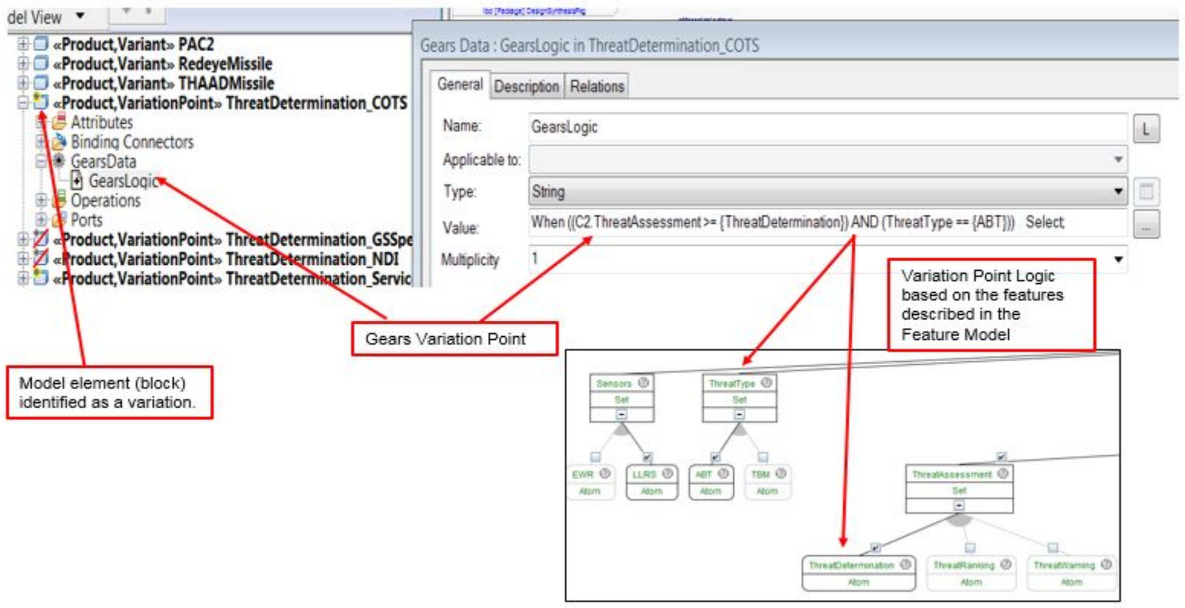


Figure 8: Feature Profile Selecting Threat Determination, Threat Warning, and Threat Ranking (excerpt)



del View

«Product,Variant» PAC2
«Product,Variant» RedeyeMissile
«Product,Variant» THAADMissile
«Product,VariationPoint» ThreatDetermination_COTS
Attributes
Binding Connectors
GearsData
GearsLogic
Operations
Ports
«Product,VariationPoint» ThreatDetermination_GSSpe
«Product,VariationPoint» ThreatDetermination_NDI
«Product,VariationPoint» ThreatDetermination_Servic

Gears Data : GearsLogic in ThreatDetermination_COTS

General | Description | Relations

Name: GearsLogic
Applicable to:
Type: String
Value: When ((C2.ThreatAssessment >= {ThreatDetermination}) AND (ThreatType == {ABT})) Select;
Multiplicity: 1

Figure 9: Rhapsody View of Gears Variation Points

In the Rhapsody model, when Gears produced the Threat Determination ABT product configuration (Figures 10 and 11), the model elements with the gear icon are either highlighted in yellow if included, or a red slash if excluded, from the projection. All other model elements not tagged with variation points are common and will always be included in the projection.

Figure 12 shows another SysML view in Rhapsody – an activity diagram – annotated with variation points for Gears to configure. A single actuation step will produce a block diagram and an activity diagram (and other diagrams we have not mentioned, for brevity) that are each configured consistently with each other to represent the same product instance.

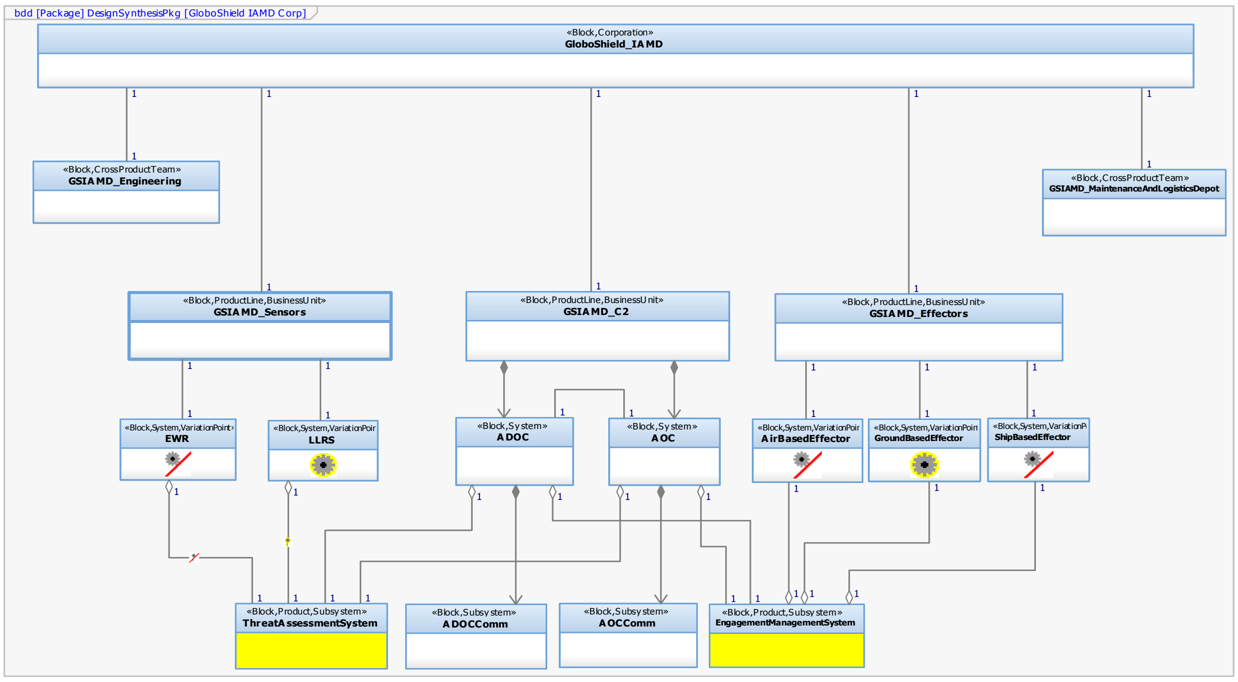

bdd [Package] DesignSynthesisPkg [GloboShield IAMD Corp]
«Block,Corporation» GloboShield_IAMD
«Block,CrossProductTeam» GSIAMD_Engineering
«Block,CrossProductTeam» GSIAMD_MaintenanceAndLogisticsDepot
«Block,ProductLine,BusinessUnit» GSIAMD_Sensors
«Block,ProductLine,BusinessUnit» GSIAMD_C2
«Block,ProductLine,BusinessUnit» GSIAMD_Effectors
«Block,System,VariationPoint» EWR
«Block,System,VariationPoint» LLRS
«Block,System» ADOC
«Block,System» AOC
«Block,System,VariationPoint» AirBasedEffector
«Block,System,VariationPoint» GroundBasedEffector
«Block,System,VariationPoint» ShipBasedEffector
«Block,Product,Subsystem» ThreatAssessmentSystem
«Block,Subsystem» ADOCComm
«Block,Subsystem» AOCComm
«Block,Product,Subsystem» EngagementManagementSystem

Figure 10: GloboShield Block Definition Diagram View After Actuation for the Threat Determination ABT Product

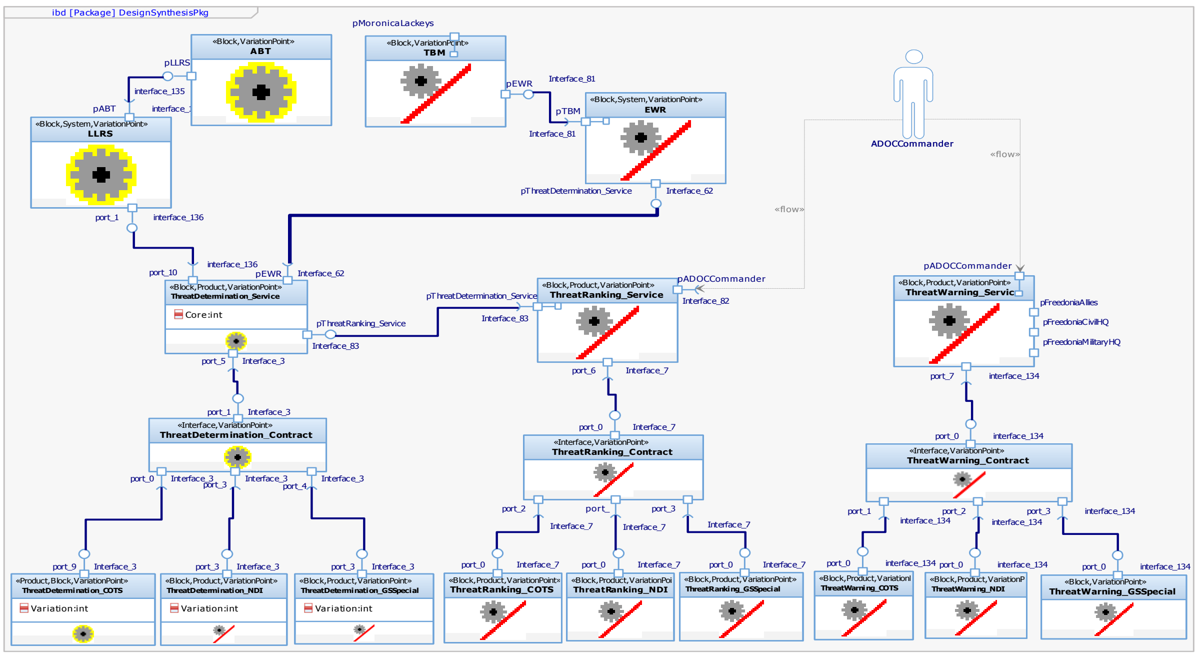

ibd [Package] DesignSynthesisPkg
pMoronicaLackeys
«Block,VariationPoint» ABT
«Block,VariationPoint» TBM
«Block,System,VariationPoint» LLRS
«Block,System,VariationPoint» EWR
ADOCCommander
«flow»
pLLRS
pABT
pEWR
pTBM
pThreatDetermination_Service
pThreatRanking_Service
pADOCCommander
pFreedoniaAllies
pFreedoniaCivilHQ
pFreedoniaMilitaryHQ
«Block,Product,VariationPoint» ThreatDetermination_Service
Core:int
«Block,Product,VariationPoint» ThreatRanking_Service
«Block,Product,VariationPoint» ThreatWarning_Service
«Interface,VariationPoint» ThreatDetermination_Contract
«Interface,VariationPoint» ThreatRanking_Contract
«Interface,VariationPoint» ThreatWarning_Contract
«Product,Block,VariationPoint» ThreatDetermination_COTS
«Block,Product,VariationPoint» ThreatDetermination_NDI
«Block,Product,VariationPoint» ThreatDetermination_GSSpecial
Variation:int
«Block,Product,VariationPoint» ThreatRanking_COTS
«Block,Product,VariationPoint» ThreatRanking_NDI
«Block,Product,VariationPoint» ThreatRanking_GSSpecial
«Block,Product,VariationPoint» ThreatWarning_COTS
«Block,Product,VariationPoint» ThreatWarning_NDI
«Block,VariationPoint» ThreatWarning_GSSpecial
interface_135
interface_136
Interface_81
Interface_62
Interface_83
Interface_82
Interface_3
Interface_7
interface_134

Figure11: Threat Assessment IBD After Actuation for the Threat Determination ABT Product

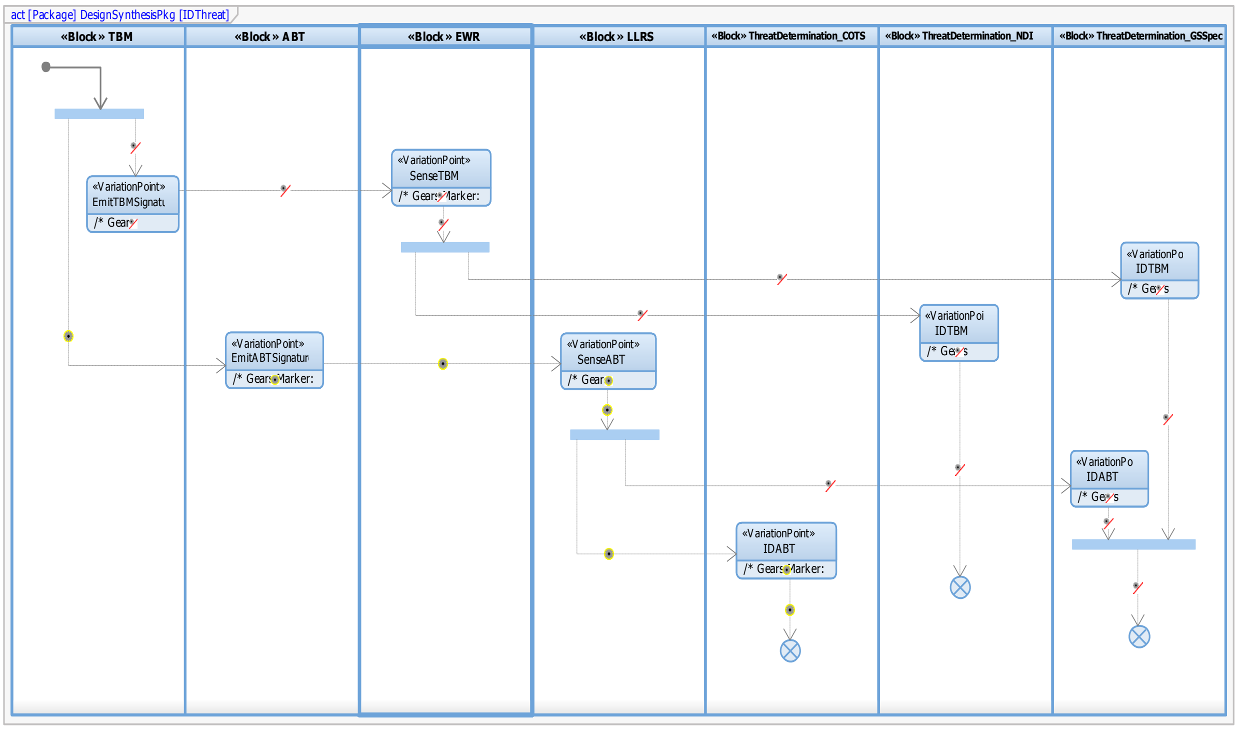
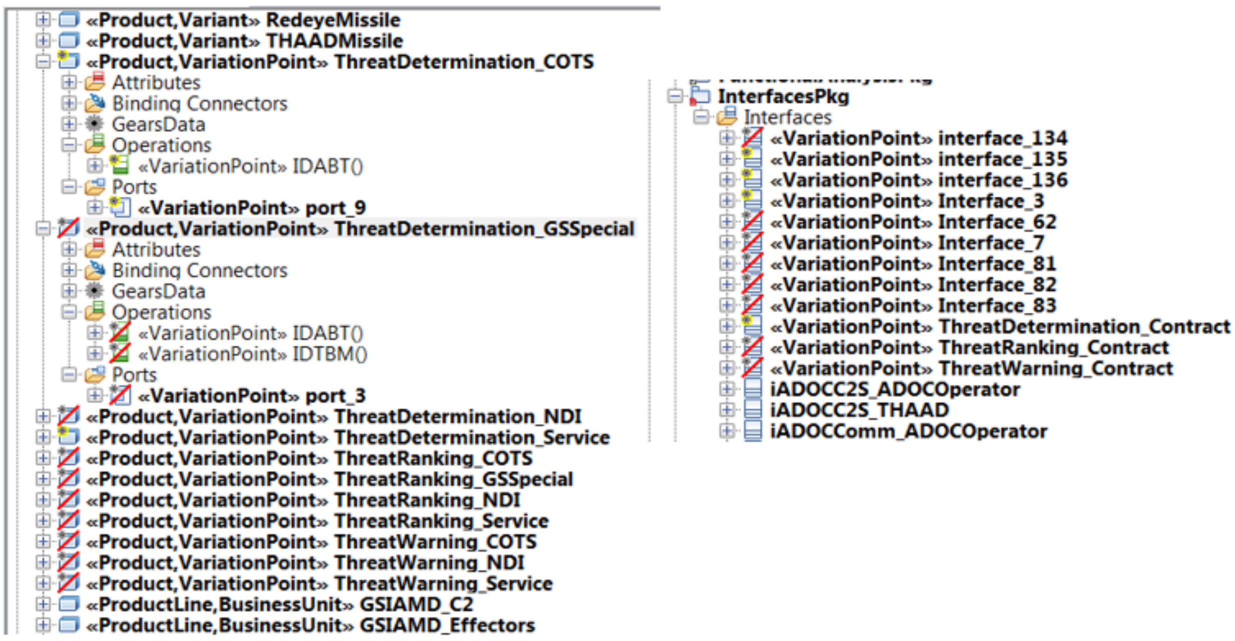

act [Package] DesignSynthesisPkg [IDThreat]
«Block» TBM
«Block» ABT
«Block» EWR
«Block» LLRS
«Block» ThreatDetermination_COTS
«Block» ThreatDetermination_NDI
«Block» ThreatDetermination_GSSpec
«VariationPoint» EmitTBMSignatu
«VariationPoint» SenseTBM
«VariationPoint» EmitABTSignatur
«VariationPoint» SenseABT
«VariationPoint» IDTBM
«VariationPoint» IDABT
/* Gears Marker:

Figure 12: Threat Assessment Activity Diagram After Actuation for the Threat Determination ABT Product

Figure 13 shows the Rhapsody browser after actuation for the Threat Determination ABT Product. The variation annotations shown within the Rhapsody browser are applied to blocks, operations, ports and interfaces.


«Product,Variant» RedeyeMissile
«Product,Variant» THAADMissile
«Product,VariationPoint» ThreatDetermination_COTS
Attributes
Binding Connectors
GearsData
Operations
«VariationPoint» IDABT()
Ports
«VariationPoint» port_9
«Product,VariationPoint» ThreatDetermination_GSSpecial
Attributes
Binding Connectors
GearsData
Operations
«VariationPoint» IDABT()
«VariationPoint» IDTBM()
Ports
«VariationPoint» port_3
«Product,VariationPoint» ThreatDetermination_NDI
«Product,VariationPoint» ThreatDetermination_Service
«Product,VariationPoint» ThreatRanking_COTS
«Product,VariationPoint» ThreatRanking_GSSpecial
«Product,VariationPoint» ThreatRanking_NDI
«Product,VariationPoint» ThreatRanking_Service
«Product,VariationPoint» ThreatWarning_COTS
«Product,VariationPoint» ThreatWarning_NDI
«Product,VariationPoint» ThreatWarning_Service
«ProductLine,BusinessUnit» GSIAMD_C2
«ProductLine,BusinessUnit» GSIAMD_Effectors

InterfacesPkg
Interfaces
«VariationPoint» interface_134
«VariationPoint» interface_135
«VariationPoint» interface_136
«VariationPoint» Interface_3
«VariationPoint» Interface_62
«VariationPoint» Interface_7
«VariationPoint» Interface_81
«VariationPoint» Interface_82
«VariationPoint» Interface_83
«VariationPoint» ThreatDetermination_Contract
«VariationPoint» ThreatRanking_Contract
«VariationPoint» ThreatWarning_Contract
iADOCC2S_ADOCOperator
iADOCC2S_THAAD
iADOCComm_ADOCOperator

Figure 13: Rhapsody Browser After Actuation for the Threat Determination ABT Product

In order to auto-generate a view of the architecture that applies only to a specific instance of a GloboShield product, the Rhapsody models can be *actuated to a staging area*. When actuated to a staging area, Gears removes all non-selected variations along with all Gears variation notation. From a staged actuated model, documentation can be generated using Rational Publishing Engine to create product specific architecture description documents and interface description documents. In addition, if the product line modeled the interfaces data, Rhapsody can generate the specific product's IDL for software. As an added benefit, the staged behavior views provide and communicate model-based detailed product specific test case scenarios that are used by the systems and software integration test teams.

## Conclusions

Raytheon has moved from generating architecture documentation from a document-centered approach to a model-based approach. Productivity gains resulted from using a model-based tool for creating product line architecture through reusable components. However, using model-based engineering alone did not relieve the problem of having to "clone and own" the variant products. The master architecture provided a framework for reusability but required the generation and maintenance of separate product architectures. This led to the approach of combining a PLE Factory concept with the MBE approach, which obviated the need to maintain separate models.

This paper has shown a simple and effective way to combine two powerful engineering paradigms, MBE and PLE. The combination is fully supported by off-the-shelf tooling and automation, all of which is in widespread use today.

The combined paradigm uses the PLE Factory concept of a shared asset superset with variation points, automatically configured to produce product-specific instances.

For MBE, architectural models in Rhapsody are the shared asset we have focused on. (We also use Gears to configure requirements, source code, and more, but this is beyond the scope of this paper.) Variation points in Rhapsody models denote model elements (e.g., a block in a block diagram) that should be included or omitted from a product instance. Thus, we have cleanly transitioned from the documentation-based realm of Figures 3 and 4 to a fully-automated model-based realm.

When our PLE factory, automated by Gears, produces a product-specific architectural model in Rhapsody, we can then carry out all of our MBE-based analysis and downstream transformations. The derivation of the architecture(s) of Figure 4 from the "master" architecture of Figure 3 is now fully automated, happens in a few seconds, and is not prone to the errors of manual derivation. Moreover, we have only the superset to store, maintain, and update. The derived architectures are not maintained on their own, but merely re-generated when the superset changes. Thus, we have cut the artifacts we need to manage and store by 80% (from 1+4 to 1).

Using a model based engineering approach to capture the superset architecture for a product line provides the benefits of capturing the architecture information and design decisions in a model repository as a single source of truth. Within the model, different views are created to provide a stakeholder the information they are concerned with. In this way, the model contains the master architectural components; the various views express the variant architectural choices for each product.

Using the PLE factory approach, provides the benefit of auto-generating product specific architecture models without resorting to clone and own techniques. In addition, product architecture documentation, interface specifications, and IDL can be generated from the staged product architectures and delivered to the end customer along with auto-generated requirements specifications that reflect the same variations.

PLE and MBE have, on their own, each reached industrial levels of maturity, backed up by solid and robust technologies and methodologies that work at large scales. We hope to have shown that MBE and PLE together – *Model-Based Product Line Engineering*, if you will – has now arrived on the scene fully formed and benefitting from the maturity of each of its parents, and providing the benefits of both.

## References

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

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

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

[4] Cohn, M. *Succeeding with Agile*, Addison Wesley, 2009.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] Rico, D. F., "What is the return of investment (ROI) of agile methods?" http://www.afei.org/WorkingGroups/ADAPT/Documents/rico08a[1].pdf, 2008.

[13] Scaled Agile Framework, www.scaledagileframework.com

[14] Software Engineering Institute, "Benefits and Costs of a Product Line," http://www.sei.cmu.edu/productlines/frame_report/benefits.costs.htm

[15] Software Engineering Institute, "Catalog of Software Product Lines," http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm

[16] SPLC Product Line Hall of Fame, http://splc.net/fame.html

[17] Wikipedia, "Scrum (software development)," https://en.wikipedia.org/wiki/Scrum_(software_development)

## Biography

**Dr. Bobbi Young** is a systems engineer and certified architect at Raytheon. She currently leads an Internal Research and Development Project focusing on adoption of PLE across the business. She is regarded throughout Raytheon as an expert in MBSE and co-chairs an MBSE Technical Interchange Group. Bobbi is also a faculty member of Worcester Polytechnic Institute as an MBSE instructor and has co-authored a book on object oriented analysis and design. She is a US Navy Commander (ret).

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