Extended Generic Product Structure: An Information Model for Representing Product Families

An information model is presented that supports sharing of design definition data between the designs of completely configured variants within a product family. Design data sharing is supported across many levels of a design’s product structure hierarchy: A change in one subassembly component does not force the whole subassembly to be duplicated. This is achieved for completely configured models and does not require the use of effectivity or any other filtering mechanism. The key is recognizing a product structure architecture that acts as a template for product variants, maximizing data sharing between them. This approach is applied to many distinct product structure abstractions, including the geometric design and the logical systems design of a product. It is extended to include secondary product structure data such as interface connection points (e.g., ports) and connectivity information, which may involve connections between ports or the mapping from the logical systems design to the geometric design that implements it. This model achieves data scalability for hierarchical product structures, meaning that when adding a new product variant, the amount of new data that must be added is proportional to the amount of design change required for the new variant times the logarithm of the total system size (this logarithm is taken to the base of the branching factor of the product structure tree). [DOI: 10.1115/1.2218361]

Keywords: product lifecycle management, multivariant design, configuration management, product structure, design domains, ports

1 Introduction

Anyone who has worked on the design of complex products with high variability is aware of the challenges involved in capturing and managing their designs: Variations of computer aided design (CAD) and system designs cannot be captured without redundancy, and minor modifications often mean copying far more data than just the changes. Duplicated data frequently means time-consuming data reassociation work for design teams and escalating costs for information systems.

The problems of computer sensible representations for complex product families must be addressed at all levels of the design process. To date, the geometric CAD domain is the most mature in this regard, but scalable, complete solutions to capture and manage common CAD/geometric data, without unnecessary duplication, are still not available. In areas such as manufacturing planning, on one end of the abstraction spectrum, and systems designs, on the other, progress has been even slower.

This paper offers a solution that focuses on capturing reusable and non-reusable design definitions and the hierarchical product design structures composed from them. We propose a product data representation that allows common design content to be shared among individual configurations of a product family. Unified modeling language (UML) class modeling is used to document this representational pattern as a metamodel that supports sharing hierarchical, reusable design components in the context of a product family. This model offers a solution to some of the key scalability roadblocks that must be overcome to manage high-variability, complex, product families.

We call the solution extended generic product structure (EGPS). It is an extension of its predecessor, generic product structure [1,2], which addresses the problem of part selection for highly variable designs by introducing the concept of a product architecture. However, it does not address (1) capturing specific assembly configurations that share design content without having to apply a filtering operation, and (2) representing instantiable, reusable design architectures.

Specifically, EGPS provides a mechanism to support the following:

- Capture precise content in individual configurations without requiring application of effectivity or option-based filters
- Share content between specific assembly configurations
- Allow product structure architectures to be defined either as in-place components, or as reusable design definitions that can be instantiated as components of higher-level architectures
- Provide a structure within which configurations can be automatically generated based on option or effectivity filtering mechanisms
- Provide a base structure from which all product design data can be captured and shared among variant configurations using a common uniform configuration mechanism

Because it is somewhat subtle, we would like to emphasize one point from above. Product families often have many more configurations (e.g., billions) than can be explicitly modeled. This has led to approaches that abandon the explicit modeling of configurations in favor of approaches that place Boolean expressions on individual design entities, in terms of Boolean variables that can relate to particular product end items (effectivity) or optional fea-

---

Contributed by the Engineering Informatics (EIX) Committee of ASME for publication in the Journal of Computing and Information Science in Engineering. Manuscript received May 14, 2004; final manuscript received May 1, 2006. Editor: R. Rangan.
tures (options). Configurations are then projected out of what is called a “maximum configuration product structure.” It is like representing all designs in one assembly and filtering out all those components that do not contribute to the one being viewed. The problem is that changes to any of those filtering variables can prevent the user from regenerating a previously generated configuration. While we certainly concur that modeling billions of configurations is not even possible and so such projection methods are necessary, we also believe that it is necessary to explicitly capture generated configurations. This paper offers a method to do just that, in a highly efficient manner that increases the database size in much closer proportion to the actual changes made to a particular configuration, as compared with current approaches. In a future publication we will address modeling effectivity and option-based configuration generation as an application of EGPS. At the other end of the complexity spectrum, EGPS enables the manual creation of a small product family by explicitly capturing the shared design content of the family members, avoiding the need for either filtering logic or managing redundant design data.

The information pattern for EGPS is complex, and so we believe the task of implementing it in an accessible, automated, design-authoring tool is essential to support the design of complex products. EGPS is represented as a set of class objects whose structure and behavior form the core of a conceptual model aimed at supporting the implementation of automated and semi-automated computer aided engineering (CAE) applications, including mechanical CAD and systems design applications.

2 Prior Art

There is a vast body of research addressing design and the representation of product data. Relatively little work has been done that addresses the ability to capture a very complete product representation while simultaneously allowing distinct design variants to share design data with each other. There is a general recognition that geometric CAD data for individual parts must not be copied due to their size and complexity, and so the ability to define multiple instances of part geometry is relatively common in CAD systems. However, CAD assembly data being much smaller in size is seldom considered in the need to share design data between design variations. For complex designs, this need is critical to understand and manage these designs.

This area of research is highly cross disciplinary, making it difficult for researchers who work on it to find each others’ work. In addition, developments in this area are often closely held by CAD and product lifecycle management companies. For both of these reasons, we apologize in advance if we have missed work that should have been cited here.

The initial recognition of the need to share complex product structure data between product variants occurred in the community that manages bills of materials for complex products. Van Veen [2] gives a clear and compelling vision of how to capture a higher level abstraction of a bill of materials by recognizing a product family architecture as a structure that can be used to enhance sharing of certain product data between members, or configurations, of a product family.

Erens [1] generalized the generic bill of materials concepts to the idea of a generic product structure that may be applied to multiple domains, each representing a product from a different perspective, and at a different level of abstraction, e.g., a technology domain representing systems to be built in a certain technology, versus a physical domain that captures a product’s geometric part and assembly design definitions.

Their work and all other previous work, of which we are aware, is at a higher level of abstraction than the logical data model presented here, which is focused on creating a conceptual object model that will be useful for implementing CAE applications and their supporting data management systems. For example, Erens [1] talks about reusable generic product structures without discussing a mechanism for representing that concept. We believe that our work complements the work of others by providing a sufficiently complete model to underpin implementation of the concepts.

Savaard [3] offers an excellent review of current approaches to product family modeling with a focus on the generation of specific configurations by application of design rules. While such work is necessary to the advancement of product family modeling, we believe that our complementary work to scalably capture the results of such generation is necessary for a couple of reasons: (1) Other information may need to be associated with the results of generating the design, but the relationships may not be able to be automatically established without human intervention. (2) When it is not possible to generate design configurations in a fully automated manner, it will be required to explicitly capture those portions of design configurations that must be authored manually.

A critical aspect of our model is that we address mechanisms for representing and sharing connectivity information in a hierarchical structure. Using ports to formally represent connection points has been addressed in the literature. This work evolved from Heisserman and Mattikalli [4], and we have generalized and extended their work to include reusable (instantiable) port definitions, hierarchical port definitions, and the ability to have distinct port configurations share content with each other.

It is interesting to note that the next generation of UML, called UML2 [5], has added an explicit composition/instancing model that is a subset of the one introduced in Section 4.1.1: see also Bock [6]. Their approach is different than this work in that it uses classes to directly represent assemblies, whereas we define a metamodel, object instances of which are used to model assembly structures.

3 A Word About “Instance”

Before going forward it is necessary to point out a language issue regarding the word “instance.” In class modeling there is a distinction between a class definition that may be described in UML class diagrams, instance of such a class might be created in a database or the memory of a computer, or be shown in UML instance diagrams. Since our data models deal with a completely different sort of instancing, we reserve the terms object or class instance for the UML/programming meaning of instance.

Otherwise instance is used to describe the idea of creating an instance of one definition as a child component of a parent, assembly design definition. This may be easier to understand when talking about geometric designs, such as creating two instances of a jet engine design on an airplane design, but it is applicable to any case where it makes sense to have multiple instances of one definition as components of others.

This type of assembly/part instantiation is an operation that has been implemented in many different CAD systems and is well known to the state of the art as a solution to the problem of managing a common definition that is instantiated in many places. Instantiating an assembly creates an instance tree whose structure is identical to the original assembly and which is linked back to it. Changes can thus be automatically propagated from a reference assembly definition to all of its instances. See Ref. [7].

We will illustrate instantiation along with other concepts underlying EGPS by example using class instance diagrams before formalizing them in the UML class model of Appendix A. Because of the confusion when using the term instance, none of the EGPS object classes have that term in their name.

4 Introducing Extended Generic Product Structure Using Class Instance Diagrams

We begin with a simple example of a six brick assembly design that could be made with building blocks. This example provides an intuitively simple way to introduce the basic terminology and describe the concepts to support capturing reusable product family designs.
4.1 Single Variant Product Structure. The six brick assembly designs described in this section use a proper subset of EGPS, called single variant product structure (SVPS), which will be introduced first and followed by an introduction of the rest of the basic EGPS structure in Sec. 4.2.

4.1.1 SVPS Classes. The fundamental class is definition. A definition object can represent the design of a single indivisible unit or an assembly of arbitrary complexity. When a definition object represents an assembly, it is the root of a definition tree that has a child node for each component. If the definition object represents an indivisible entity, it simply does not have any components.

A definition object must be in one of two states: reusable or in place.

- A reusable definition is a root definition, and is not a component of any parent definition. Instances of it can be created as child components of other definitions.
- An in-place definition is not a root definition, which means it is a component of a parent definition. An instance of it cannot be created as a component of another definition.

Usage is a class used to define assembly instances of reusable definitions. A usage object must be a component of a parent definition, which may be either reusable or in place.

4.1.2 Understanding Single Variant Product Structure by Example. We now present a series of assembly designs for an assembly of six bricks. These designs are illustrated using class instance diagrams. The objects in Fig. 1 are class instances of the definition class and the usage class. We begin with a simple, hierarchical structure.

In Fig. 1, the reusable assembly definition, SixBrick, is composed of three in-place definitions: Top Two Brick, Middle Two Brick, and Lower Two Brick, each with its own independent left and right usage components. All six usages are instances of the reusable definition, Brick, as captured by the instance links.

While this design accurately captures the SixBrick assembly, it has a couple of shortcomings: First, the design intent—to have an identical definition (including part positioning and attributes) for the Middle and Lower brick assemblies—is not enforced by the design. Since the left and right usages in Middle Two Brick and Lower Two Brick are all independently authored, their relative positions can be different in one assembly versus the other. The second issue is poor scalability of the design when the number of bricks in each subassembly is increased.

Since the new design of Fig. 2 involves creating instances of an
assembly, one more class, *occurrence*, is introduced. Occurrence objects are shown as octagons in the example designs. Occurrences are similar to usages because they represent instances of other assembly components. The difference is that they are derived, rather than directly created by a designer. When a usage is created to represent an instance of an assembly, occurrences are automatically created to represent instances of all the components of that assembly.

In Fig. 2, we add a reusable assembly, TwoBrick, to solve the design-intent and scalability issues.

In Fig. 2, SixBrick is still decomposed into three components. Top Two Brick is the same in-place design as in Fig. 1. The Middle and Lower usages are instances of the reusable definition TwoBrick, replacing in-place definitions Middle Two Brick and Lower Two Brick from Fig. 1. TwoBrick uniquely defines the relative positions of the Left and Right occurrences for both the Middle and Lower TwoBrick usages.

This first shortcoming of the previous design is removed by eliminating the ambiguity of the relative positions of two bricks. The second shortcoming is apparent if the number of bricks in each assembly was greatly increased, in which case this second approach would greatly reduce the number of design objects the user would have to explicitly create.

4.1.3 Designing Product Variants Using SVPS. Next, we use SVPS to capture the design of a SixBrick product family with two design variations: First, the assembly of six identical bricks from Fig. 2; and second, an assembly where the left middle brick has a spy hole in it.

In Fig. 3, we model this family by renaming the SixBrick assembly 6B-A. We also rename TwoBrick to 2B-A, and Brick to 1B-NH. A second brick definition, 1B-H, with a spy hole, is added. Then 2B-B is added, a second two brick assembly with one spy hole, is added. Finally, we add the variant six brick assembly, 6B-B, with the spy hole.

Figure 3 has two distinct SixBrick assembly definition trees, 6B-A and 6B-B, each with an in-place definition and two instances (usages) of either 2B-A or 2B-B.

This SVPS model does not allow sharing of common components between these two SixBrick variations, despite the fact that 6B-A and 6B-B actually have two common components, Top Two Brick and Lower, and only one distinct component, Middle. All three components are represented by distinct objects in each assembly. Only the reference definitions, 2B-A and 2B-B, are shared between the two.

Since the identical components Top Two Brick and Lower are represented by distinct objects, it is ambiguous as to whether, from a computer-sensible standpoint, they actually have identical meaning. Ironically, this duplication of design tree components reduces information content, while increasing the design data size.

Note that we have intentionally modeled Top Two Brick as an in-place definition. This is because there are times when an assembly will only be used once per product and is specialized to a particular product family. In this case, we believe it is a bad idea to represent this as a reusable definition since that implies that it makes sense to instantiate it multiple times on one assembly or on multiple parent assemblies. In the next section, we introduce an approach that allows such an in-place design to be shared between product variants without implying that it is a reusable design. Finally, it is interesting to note that most current CAD systems would enforce this data duplication.

4.2 Introducing Extended Generic Product Structure (EGPS). We now introduce the EGPS model by extending the SVPS model to maximize design data sharing among product variants.

4.2.1 EGPS Classes. EGPS adds three new classes, each a configuration class of the previously introduced SVPS classes: definition, usage, and occurrence. We refer to the SVPS classes collectively as *Master Classes* and refer to the new ones as *Configuration Classes*. They are definition configuration, usage configuration, and occurrence configuration, respectively. Each configuration class object represents a configuration of one-and-only-one master class object: the configuration objects each implement their master object, and are a configuration (or variation) of it. Their behavior is directly analogous to their master’s.

Just like a definition, a definition configuration object has two states:

- It can be a reusable definition configuration whose master must be a reusable definition
- It can be an in-place definition configuration whose master must be an in-place definition. It may be included as
a component of multiple, parent definition configurations but cannot be instantiated by a usage configuration.¹

There are two classes used to represent instances of reusable definition configuration trees:

- A definition usage configuration is a configuration of its master usage. It may be included as a child of multiple definition configurations.
- An occurrence configuration is a configuration of its master occurrence. It may be included as a child of multiple usage configurations or occurrence configurations.

Just like an occurrence, it is a derived object, created automatically when a usage configuration instantiates an assembly.

Each configuration object is subordinate to its master object. If the master object is deleted, the configuration object must also be deleted. Thus, a master object owns its configurations. The inclusion of usage configurations, occurrence configurations, and in-place design configurations in multiple parent definition configurations is the mechanism that enables design content sharing between configurations.

4.2.2 Understanding EGPS by example. We now revisit the two variations of our SixBrick product family. We demonstrate EGPS captures the same two product variations in the following series of logical steps. In our EGPS instance diagrams that follow, the configuration objects are shown as contained within their master objects to avoid drawing more links.

4.2.2.1 Step 1: Create the master definition trees. In Fig. 4, we reinterpret the SVPS design of Fig. 2 as a product family architecture in which it acts as a template that must be filled in to create a configuration of the SixBrick assembly design. Note that the single variant design of Fig. 2 is still a valid EGPS design.

4.2.2.2 Step 2: Create a configuration that is a copy of the original assembly. In Fig. 4, we add a configuration object to every master object in the SixBrick assembly design. The assembly names introduced in Fig. 3 are used for the configuration objects. The master object names are the SixBrick assembly names of Fig. 2. The configuration object names are the names from the 6B-A assembly of Fig. 3. This creates two parallel assembly structures, with the configuration trees imbedded in their corresponding master trees.

In Figs 4–6, we show letters in the usage configuration objects instead of an explicit link to the definition configuration objects they reference. For example, the usage configuration in the Middle usage, has the letter A which indicates that it is an instance of the 2B-A definition configuration of the TwoBrick assembly. This shorthand is unambiguous because each usage configuration (e.g., A) must instantiate a definition configuration (e.g., 2B-A) of the definition (e.g., TwoBrick) instantiated by its master usage (e.g., Middle).

4.2.2.3 Step 3: Add a lightweight copy of 6B-A, and a variant of TwoBrick and Brick. In Fig. 5, we make a lightweight copy of the 6B-A assembly configuration. Independently, we add a spy hole configuration to both Brick, called 1B-H, and to TwoBrick, called 2B-B.

Only the 6B-2A object was added to create a copy of 6B-A. Both copies share all of their components, including the Top Two Brick in-place definition configuration, TB. This sharing of TB allows sharing of design data without forcing the creation of a reusable assembly, and a usage of it. The 2B-B spy hole configuration of TwoBrick shares the Right usage configuration with 2B-A and adds a new Left usage configuration instantiating the spy hole brick, 1B-H.

4.2.2.4 Step 4: Rename 6B-2A and add a new configuration of usage, Middle. In Fig. 6, 6B-2A from Fig. 3 is renamed to 6B-B and connected to the new Middle spy hole usage configuration, B, which instantiates 2B-B, thereby adding a spy hole brick occurrence to the 6B-B configuration. 6B-B maintains its component configuration links to the shared component configurations in Middle and Top Two Brick.

4.3 Benefits of Using EGPS Versus SVPS. EGPS minimizes the number of objects used to capture design variations, while still capturing the design content explicitly for each. Figure 6 shows one efficient and scalable design of the two SixBrick assembly design variations by recognizing the SVPS definition tree of Fig. 2 as a template assembly, which provides a mechanism for sharing all common design data between the two design variants. The fact that five out of six elements are the same between configurations 6B-A and 6B-B is captured precisely by the fact that their individual definition trees share five out of six leaf objects.
5 Design Domains and Primary Class Specialization

We now discuss the application of EGPS to representing the different levels of abstraction that are necessary to represent the design of a complex product or product family. For simple designs, like SixBrick, modeling six parts in a global context may be acceptable, and EGPS may be overkill. For complex designs, however, teams must work with the product design at multiple levels of abstraction.

5.1 Granular Abstraction. One type of abstraction deals with granularity. The design is represented, at the coarsest product level, by a top assembly node and its next level component assemblies. At this level, an airplane may be viewed as composed of wings, engines, tail, etc. Each of those components can be decomposed until you get to the finest level at which an entire product can be viewed as a composition of instances of indivisible parts. At this level of granularity, the same airplane would be viewed as being composed of ribs, tubes, rivets, etc. The granular abstractions are naturally captured by levels within an EGPS product structure.

5.2 Design Domains. Another type of abstraction considers that the entire product can be described from different perspectives. For example, SixBrick is a design of the geometry, but not, for example, in what sequence it should be built. For highly complex products, there are multiple perspectives from which design teams must represent and view the product. These perspectives typically span a range, from functions and simulations, to physical shape, to manufacturing plans. We define the concept of a design domain, for each such perspective, which allows us to simultaneously represent the product design from multiple, distinct perspectives. Design teams must decide which design domains are useful for a given product. There is not one correct set of domains. In fact, there are complex trade-offs involved in the decision of whether or not to introduce a new domain. Experience has shown that, in many cases, forcing multiple design perspectives into a single domain can cause much higher design and maintenance effort than having multiple domains. It is critical to distinguish multiple design domains from multiple views that can be projected from one design domain.

Fig. 5 Adding a lightweight copy

Fig. 6 Two variants are completed
Here we describe three domains that have proven useful in capturing complex product family designs. They follow, listed from higher to lower level of abstraction:

1. **System**
   This represents logical system elements, logical interfaces, and connectivity structure between those elements, without describing their mechanical embodiment. System assemblies are often not realized by one physical assembly because their purpose is to clearly describe the elements and connectivity of a given system, not its geometric composition.

2. **Geometric**
   This represents the part geometry and physical specifications, including interfaces. It also captures the geometric assembly structure, which includes the definition of where to position all parts and all assemblies at each level in the product structure. Aside from complexity management, the primary purpose of assemblies in the Geometric Domain is to support reusable design definitions; the secondary purpose is to efficiently support mappings from the System Domain to the Manufacturing Domain. This domain’s product structure includes the traditional parts list (bill of materials) as a projectable view.

3. **Manufacturing**
   This represents the sequence of assembly steps in which the product is constructed, and the factory installation locations. The assembly sequence can be as simple as putting together assemblies defined in the Geometric Domain, or, for more complex, built-up products, the assembly sequence can be decomposed into groups of operations, performed by a mechanic, that install groups of part instances in a certain period of time. This lowest level of manufacturing assembly is sometimes called a job. Jobs are collected into higher level manufacturing assemblies that are further collected, based on the factory location where the parts are installed. Reusable manufacturing assemblies enable a complex assembly operation to be easily relocated from one place to another.

Other important domains include Functional Design, System Simulations, Product Support and Maintenance. Ref. [1] recognizes three domains: Functional, Technological, and Physical. The last two domains correspond, roughly, to our System and Geometric Domains, respectively. A future publication will deal with an end-to-end representation, including the Functional Domain.

Each design domain describes the product design from a certain level of abstraction, and acts as a requirement for the next less abstract design domain. This means that, except for data that are fully specified in a given domain, each component in one domain must be implemented by components in the next, less abstract domain. A complete design must capture this accountability mapping between such pairs of domains. See Ref. [8] for a discussion of how an accountability mapping can be established between single variant design domains.

### 5.3 Class specialization

Every domain has an EGPS product structure (assembly) for each reusable design defined in the domain. They are defined using specializations of the three EGPS Base Master Classes (definition, usage, and occurrence) and three EGPS Base Configuration Classes (definition configuration, usage configuration, and occurrence configuration). For each distinct domain, these six specialized subclasses are called the Primary Classes of that domain. A domain’s Primary Class objects are used to form the product structures of the domain. The subclass names for any specialization of the EGPS Master Classes are derived by combining the base names and the name of the class that inherits from definition. The master classes of the System Domain are system, system usage, and system occurrence. The configuration class names are always derived by adding “configuration” to the end of the Master Class names, e.g., system configuration. The definition name for the Manufacturing Domain is sequence, and for the Geometric Domain is part.

At this point we need to acknowledge that using the abstract EGPS Base Classes in the introductory material for Sixbrick was not technically correct. A geometric specialization of the EGPS Base Classes—part, part usage, and part occurrence—should have been used instead of definition, usage, and occurrence. A similar substitution should have been made for the configuration classes. We intentionally chose that path in order to introduce the basic vocabulary of the EGPS pattern using a specific, geometric example.

### 6 Secondary Classes: Ports and Associations

As explained, **EGPS Primary Classes** are used to define the primary structure of a design domain. Here, we extend the idea of product structure to include what we call **EGPS Secondary Classes** that capture secondary aspects of the design in a given domain. They include elements such as physical ports on parts, or logical ports on systems.

Here we focus on two important secondary classes:

1. **Port Classes**, which establish and enable formal management of specific connection points on the primary product structure objects. These are important in managing interfaces between primary components.
2. **Association Classes**, which capture all relationships that are not modeled explicitly as class relationships.

Note that even attributes could be treated as such sharable, secondary product structure elements.

#### 6.1 Ports

Ports are a particularly important EGPS Secondary Class that identify distinct connection points on a primary definition and can contain distinct attributes for them.

Think of an instance of a hydraulic pump that has two connections to an instance of a hydraulic actuator: one connection for pressure, another for a return line. This is captured in the Systems Domain by connecting each of two port occurrences on the pump instance to a corresponding port occurrence on the actuator instance.

For complex systems, with many ports, hierarchical EGPS behaviors are necessary to manage complexity. Even though a port class is an EGPS Secondary Class, it can be defined as a reusable port definition to allow a common port structure to be defined and instantiated as a component of multiple primary definitions. This is achieved by creating a port usage to instantiate the reusable port definition as a component of a primary definition. A set of reusable ports could be organized in a port library.

EGPS Port Classes play one of three roles: (1) An external connection point definition that is a component of a reusable definition. (2) A stand-alone reusable port definition. (3) A connectable port on a definition tree component. An implementation would establish one base port secondary class from which other connection point classes could inherit. The behavior of such a base class is addressed next.

---

2The system structure has sometimes been called the physical design by the systems engineering community, because it is more physical than the functional design. Others call the geometric design the physical design. That is why we avoid naming a physical domain.

3An example of data that are in their final form in the System Domain is the data that capture the location of software modules in the system and track the exchange of variables between them. Therefore, such objects would have no mapping to the Geometric Domain.

---

4A class relationship is formally modeled as a relationship between the definitions of the two classes. (See Appendix A.1.)
Any port whose primary parent is a reusable design is an external connection point definition. These cannot be connected to any other ports. When ports have primary parents that are components in a definition tree, they are connectable. Within a common definition tree, the association of two connectable ports represents a connection between the primary parents of those ports. Connectable ports can be connected to each other from different levels in a definition tree, except when their primary parents have a direct or an indirect parent-child relationship. This means that a port on a child component may not connect to a port on its primary parent or any other up-tree parent. Such a relationship is only valid as a promotion, which will be addressed below.

6.2 Associations. All association objects relate two other objects to each other. Hierarchical assembly, instancing, and exclusion relationships are not captured with association objects, because they are modeled explicitly as class relationships. Associations must be in-place, and must have a direct or indirect parent primary definition. This means that they are defined in the context of a particular definition tree, and may only relate other objects that are also in that tree. Occurrences of them may be used to relate objects in the tree to which the occurrence belongs.

6.3 Connection and Promotion Associations. Two significant local association classes are connection and promotion. Connection objects establish a connection between connectable ports that are in a common definition tree. Promotion objects link an internal connectable port to an external connection port, on the root or an in-place definition. The internal port is said to be published on the external connection port. Each instance of a system has a connectable instance of each external port that is defined on that system. Some systems, called "black boxes," require that any connection to an occurrence of one of their internal components must be made via this promotion mechanism.

6.4 Business rules. As previously mentioned, there are many constraints on the relationships that are specified by the EGPS Base Class model. In general, parent-child relationship business rules must be specified for each Primary and Secondary Class subtype. Some of them are intuitively obvious: e.g., A port cannot be a child of a port. A system cannot be a child of a port.

Primary classes like systems and parts usually have the full EGPS behavior, allowing both in-place and reusable definitions. In addition, business rules may be applied to limit a certain secondary definition subclass to only allow in-place definitions. These types of business rules may vary between or within companies, reflecting different policies or useful product abstractions, given certain product types or business processes. A company may choose to model their products with no System Domain at all.

In addition to parent-child rules, there are also connectivity and general association relationship rules. These rules depend upon the Primary Class type of their definition tree, what their object types are, and in some cases their particular attribute values. For example, port connectivity may depend on type and directionality.

6.5 Connections in the System Domain. In the System Domain, a connection object is used to associate two connectable system ports. The System Domain captures the connectivity requirements for the geometric design, so the meaning of a system connection object is as follows:

A primary system component and its ports will typically be implemented by an instance of a geometric part, which must have geometric ports corresponding to the system component’s logical ports. The connection object acts as a requirement that the corresponding ports in the Geometric Domain must be geometrically connected to each other.

Usually system designs require a net between the equipment entities, reflecting the fact that system equipment is usually connected by a geometric transport element, like a tube or wire, and not directly attached to each other. Such a system net is a primary, system definition component which must be implemented in the Geometric Domain by a transport element.

7 System Design Examples and Projections

The following examples introduce hierarchical, multivariant ports, nets, connections, and promotions for a simple system design. We also use these examples to introduce the concept of a projection that is a single variant view of a particular definition configuration. Such a view is critical for implementing authoring tools, so designers do not have to deal with the full complexity of

Fig. 7 Legend

Fig. 8 Generator with ports

Fig. 9 Multivariant generator with ports
the EGPS model as they view and edit their designs. Effective implementation of EGPS requires such projections.

The following examples present a system design: first of a generator system with ports, second, of an engine system that includes instances of the generator, and finally by adding port and connectivity information to the engine system design. The legend in Fig. 7 serves for these examples.

7.1 Generator With Ports. The Generator System of Fig. 8 has an in-place port, P1, with three in-place port components: Control, Power1, and Power2. To visually distinguish P1 from an in-place system, the parent-child link between it and the Generator System is drawn from the side of the generator. P1 represents a requirement for a physical connector on the geometric design that implements the Generator System. The three in-place component ports are requirements for three pins within that connector.

Figure 9 converts the single variant design of Fig. 8 to a multivariant design with two configurations: G-1 and G-2. P1 also converts to multivariant with two configurations: c1 and c2, which differ by P1:c1’s exclusion of Power2:c1.

Configuration names are always scoped to their master object. Therefore, the four in-place definition configurations called c1 are independent of each other. In referring to a configuration, the master object’s name can be prepended to the configuration object’s name, as in the previous paragraph. It is also true that all in-place definition and usage names are scoped to their parent assemblies.

So, if there were two ports called P1, one on system S1 and the other on system S2, then they could be unambiguously addressed by S1:P1 and S2:P1.

The idea is that G-1 is a standard unit with one power output port, and G-2 is a high power unit with two power output ports. The link from P1:c1 to Power2 is called an exclude link, and indicates positively that P1:c1 should not include a configuration of Power2:c1. G-1 and G-2 have component configuration links to select their port configurations.

Figure 10 shows a single variant view of the G-1 configuration projected from the Generator System. It is created by simply beginning at G-1 and traversing the down tree configuration links, showing each configuration object encountered.

7.2 Allowing for Single Variant Components Within a Multivariant Structure. Figure 11 is the same multivariant design shown in Fig. 9, but it allows for single variant components in a multivariant architecture. In Fig. 9, each of the three leaf-level ports has only one configuration: they are single variant definitions. They were given a configuration, only because their parent was converted to multivariant. Since the configurations add no information, it is quite reasonable to let the master object play the role of both definition and definition configuration. Each child component, with no configurations, is automatically included as a child of each parent definition configuration that does not have an exclude link to that child component. If a child component has configurations, then the parent configurations must select one, and only one, of those configurations, unless there is an exclude link.

Figure 12 shows projections of the two Generator System configurations. This illustrates how the definitions Control, Power1, and Power2, are projected as components of P1’s configurations.

7.3 Engine System Example. Figure 13 shows a simple Engine System that is composed of two usages, Left and Right, that are instances of the Generator System, from the previous example.

The Engine System has two configurations: E-1 is composed of two instances of G-1. E-2 is composed of the same Right, G-1 instance, and one G-2 instance.

Figure 14 shows the complete design, in which G-1 is a normal capacity generator variant, which needs one output port for power and one for control, whereas G-2 is a high capacity variant needing two output ports for power and one for control. This means that the Engine System must have one variant with two and one with three power output ports.

Ports P1, P2, and P3 are added to the Engine System, defining its external interfaces. P3 is included in configuration E-2, but not E-1, because of the exclude link from E-1 to P3. Were the engines instantiated on a higher level product, each usage of the engine would have three port occurrence components that are connectable instances of these three system ports.

The generator design is as described in the discussion of Fig. 11. Both usages of the Generator System in Fig. 14 (Right and

---

5Not all system ports represent a requirement for a physical connector. That status could be specified by a specific port attribute. Such a port would represent, instead, the externalization of an internal port.
Five Single Variant association objects have been added to the Engine System: promotion objects Pr1, Pr2, Pr3, and connection objects C1 and C2. Each promotion object links an internal connectable port on a component of the engine system to one of its external interface ports: Left:Power1 is promoted to Engine:P1 via Pr2, Right:Power1 is promoted to P2 via Pr1, and Left:Power2 is promoted to Engine:P3 via Pr3.

Net1 is another Single Variant child of the Engine System. It represents a requirement for a transport element connecting the two control ports in the Geometric Domain. C1 and C2 connect the net ports to the control ports on each of the two generator instances.

Figure 15 shows a single variant projection of the E-2 Engine System configuration. It is interesting to note how all single variant components of the engine system (which are not excluded) are completely shared between the two variants. The projection of E-1 is analogous to E-2.

8 Future Work

Publication of completed work on several topics is our next priority, including the following: 1. Describe how drawings and diagrams (e.g., schematics) can be captured as auxiliary data, related to EGPS product structures. 2. Employing EGPS to support the generation of configurations based on effectivity and options. Model requirement and functional designs integrated with the logical system design using EGPS.

New research will focus on applying EGPS to develop lumped-model, ordinary differential equation product simulations at multiple levels of abstraction, including functional simulations, system simulations, and electromechanical simulations representing kinematic mechanisms.

Acknowledgment

I would like to thank all those who have shared their knowledge and ideas in the area of design. In particular, I would like to acknowledge Carl Pearson, Charles Erignac, and Steve Heinz who have been great collaborators in the implementation of automated design tools that use, and therefore helped to evolve, the concepts in this paper. Mark Williams, John Monahan, Bob Malone, and Dan Fogarty have all shared their extensive engineering domain knowledge and helped provide guidance about how these ideas could be most beneficially delivered to engineers. Finally, I would like to thank Karla Steele for her editorial help.
Appendix A: EGPS UML Data Model

A.1 Introduction

This section describes SVPS and EGPS data models in terms of the industry standard universal modeling language (UML) that has been widely covered by training and educational publications, including Ref. [9]. Please note that we are using UML as a meta-language specification and are not using or extending UML composition mechanisms. In Fig. 16, the boxes’ four square corners represent class definitions, with names shown at the top. Class attributes are listed in the middle region of the box.

Although class models generally have a large number of at-
tributes of different types, the data model shown in Fig. 16 is more focused on the structure of the model than in attribute particulars. That structure is captured by the associations between classes, which are shown as lines connecting two classes, or connecting a class to itself.

Each association has 3–5 labels on it. The numbers, or number ranges—such as 2 or 0..1 or 0..n—at the ends of each association line are called the “multiplicity” of that end of the association.

Consider that an association, R, has two ends: Class A and Class B. The multiplicity of end A defines the number of R associations that may be established from a Class B object to distinct Class A objects. A specific number means that a number of associations is required. The following example multiplicities imply that the listed numbers of associations are required: No multiplicity means 1: 0..2 means 0, 1 or 2; 0..n means 0, 1, up to an arbitrarily large integer number.

The end labels, formally called association ends, are preceded by a + symbol, and they indicate the “role” of that end of the association in this description. They indicate the role played by their object, from the perspective of the object on the other end. For example, a role +assembly indicates that its class instances aggregate objects at the other end of the association. The role on the other end of the association could be called +component, if the aggregation has the meaning of a parent-child assembly relationship.

The italicized name between the two roles is the association’s unique name.

There is one more special type of association, with two variations. When an association has a diamond on one end, it indicates that the object on that end aggregates objects on the other end of the association.

- Strong Aggregation: If the diamond is filled, then the aggregation is by value, which means that the aggregated objects cannot exist without the aggregating object.
- Weak Aggregation: If the diamond is empty, the aggregation is by reference, which means that the aggregated objects can exist without the aggregating object.

A class object should only be aggregated by value, by one or zero other objects.

The boxes with a folded corner tag are comment notes. They have an optional link to the classes or the associations to which the comment is addressed.

As mentioned at the beginning of the paper, we reserve the terms object or class instance for the UML/programming meaning of instance. Otherwise, “instance” is used to describe assembly instantiation of reusable definitions.

A.2 Single Variant Product Structure (SVPS) Model

The Extended Generic Product Structure data model of Fig. 16 is technically a metamodel because the class members are not themselves used capture assembly design. Instead class instances of the data model classes capture assembly designs. Therefore, much of the following narrative elucidates the data model by describing the behavior of class instances. The model of Fig. 16 contains, as a subset, the single variant product structure (SVPS) data model, comprised of classes Definition, Usage, and Occurrence, and associations Instance1–4 and Component1–4. Definition, Usage, and Occurrence are Abstract Classes, because any system implementing an authoring tool based on them must define derived classes representing a specific type of design.

Definition is the fundamental class, because primary design domain classes, like part and system, are derived from it. The other two classes represent instances of assembly components.

Definition objects may be used to define a tree using the component association. The top definition object in such a tree has no parent definition, and is a reusable/instantiable definition. All other definitions in the tree have a parent definition and are not reusable: they are called in-place definitions, and are defined only in the context of their parent assemblies.

As mentioned above, definition is considered the fundamental class, but really reusable definition class instances provide the central objects of this representation. Each such definition object defines a complete and independent definition context. Although usages of a reusable definition may be created as child components of higher level assembly definitions, the reusable definition itself is independent of all such assemblies.

Each usage2 is an authored instance of a definition, and is always a component of one, and only one parent assembly definition, as shown by the component2 association. That association also specifies that an assembly definition may have 0..n component usages. Instance2 is used to capture the association from a usage to its references definition. A definition may be instantiated by 0..n usages.

Occurrences are always components of either a usage object, via association Component3, or of another occurrence object, via association Component4. These two associations are mutually exclusive. Each occurrence represents an instance of one component in a reusable definition tree. It therefore has one of the three mutually exclusive instance associations, depending upon what it instantiates: Instance1 for an in-place definition, Instance3 for a usage, or Instance4 for an occurrence.

When using the SVPS model subset, all objects have the SingleVariant attribute set to TRUE.

A.3 Extended Generic Product Structure (EGPS) Model

To model a variation, or configuration of a product family using the EGPS model of Fig. 16, the SingleVariant attributes are set to FALSE for all product structure elements that have multiple variants. The three Configuration Classes (Definition Configuration, Usage Configuration, Occurrence Configuration) are used to model distinct configurations for all product family components that have multiple variants, or instantiate other objects that have multiple variants. We use the term multivariant design when multiple variants of a product family are being modeled. In that case, we call Definition, Usage, and Occurrence Master Classes, in contrast to the other three Configuration Classes. When developing an EGPS multivariant product structure, the Master Classes have exactly the same structural behavior as in single variant mode, but they have a different interpretation. Instead of representing the design of an individual configuration, they provide a product structural template to which individual product configurations must adhere. We call this template a product-structure architecture: it is a logical structure that constrains and defines the structure of individual configurations.

When some components of a multivariant architecture are single variant (do not have multiple configurations) then the Master Class objects also play the role of the Configuration Classes, as described in Secs. 7.2 and 7.3.

Unlike the master objects, the configuration objects may be components of multiple assemblies, as seen by the 0..n or 1..n multiplicity on the +assy ends of associations, Component5-8. This ability to share components is why the open diamond (aggregation by reference) is used on the +assy ends of those four associations. This means, for example, a child usage configuration would not have to be deleted if one or more of its parent design configurations were deleted, whereas that same usage configuration must be deleted if its master usage is deleted.

---

2This is a shorthand for the four relationships whose names begin with Component and end with a digit in the range 1–4.

3We will omit the term object when it is clear from the context that we are talking about an instance and not the class definition, itself.
Association Component6 can have a 0 multiplicity on the +assy end. This allows a usage configuration to exist without a parent assembly. This means that designers can author configurations of a usage and later specify a design configuration of which it is a component. This is particularly useful when this model is extended to include the ability to generate configurations based on option selection and/or effectivity. That will be addressed in a future publication.

Associations Implement1-3 represent the ownership relationship between the master class objects and the configuration objects that implement them. Finally, the exclusion associations, Exclusion1-2, indicate that the definition configuration should not have a component configuration that implements the component8 on the +exclude end of the association. Without any exclusion links, each definition configuration must have one, and only one component configuration from each multivariant component of its master definition. Without this type of link it would not be possible to distinguish an error of omission from a purposeful omission of a component configuration.

The exclude link is particularly important when a multivariant master class object has a single variant master component, because definition configurations automatically include such single variant components (either usages or in-place definitions) as implicit children (for use in projected views) unless the configuration has an exclude link to that component. If the component is a usage, it must be an instance of a single variant definition. The exclude link is critical for distinguishing between an incomplete design configuration and one that legitimately excludes an architectural component.

References