A Configuration Management Model for Software Product Line

Liguo Yu[1] and Srini Ramaswamy[2]

[1]Computer Science and Informatics
Indiana University South Bend
South Bend, IN 46634, USA
ligyu@iusb.edu

[2]Computer Science Department
University of Arkansas at Little Rock
Little Rock, AR 72204, USA
srini@acm.org

ABSTRACT. Software Product Line has proved to be an effective approach to benefit from software reuse. Configuration management, an integral part of any software development activity, takes on a special significance in software product line context. This is due to the special property of software product line, in which the core assets are shared by all products. In this paper, we compare the existing configuration management models and analyze the artifacts that need to be configuration managed in software product line. We then present an evolution-based configuration management model for software product line, in which, the configuration management is divided into two domains, the production domain and the product domain. In this model, the evolution propagation of corrective changes and enhancement changes on different configuration artifacts follow different paths. The advantages and the constraints of this model are also discussed.

Keywords: Software configuration management, software product line, software evolution, change management.

(Received February 2, 2006 / Accepted March 7, 2006)

1. Introduction

Software product lines [1] [2] [5] are a well-known approach in the field of software engineering. In a software product line, a set of related products are produced through the combination of reused core assets together with product-specific custom assets. Software product lines have proved to be an effective way to benefit from architecture level reuse.

Software product lines exhibit some characteristics, such as product evolution and compatibility, which can be found in assembly lines of manufacturing industry. Accordingly, they have the similar management issues, such as change control and evolution management.

Software development and maintenance are dynamic processes where software engineers constantly modify their systems. As a consequence, software systems constantly evolve. Configuration management (CM) is the control of the evolution of systems [4] [6] [7] [8] [9] [10]. It is the discipline that enables us to keep control and track software changes. Because changes and evolution are inevitable for any software system, configuration management is an integral part of any software development and maintenance activity. The activities associated with CM include configuration artifact identification, version management, release management, branch management, variant management, and change management [3]. Change management is associated with the evolution of the configuration artifact. Because software configuration artifacts are interdependent, changes to one artifact may affect the evolution of other artifacts. Usually change management is strictly controlled by authorized personnel.

Like other fields of software engineering, software product line also needs to support configuration management. However, due to the special property of software product line, it poses challenges for configuration management [16]. The special property of software product line is all the products consist of same or adapted similar core assets. The products, the core assets, the custom assets, and the components within the assets all need to be configuration managed. The challenges for configuration management in software product line include: configuration artifacts determination, evolution management, and product line "decay" prevention.

Configuration artifact determination: Software product line is also called *software product family*. There are more member products in one family than in conventional software systems. Hence, in product line, there are much more number of products, assets, and components that needs to be configuration managed. To reduce the working load and the complication of configuration management, it is important to select the right artifacts under configuration management.

Evolution management: Software product line must control the changes to all artifacts under configuration management, especially the core assets. Due to the interdependencies between the assets and the products, changes to either of them may affect the evolution of the other. On the other hand, because two products may share the same core assets, evolution of one product may also affect the other.

Product line "decay" prevention: The benefits of software product line come from the reuse of core assets. If changes to software artifacts are not well controlled, this may result the core asset deviate from the general architecture or the product loss its connection with the core assets. Both cases will decay the software product line. This is also called "erosion" [12] and "software aging" [13] in conventional software.

To address these issues in software product line, several configuration management models have been proposed [5] [11] [16]. All of them lack the capability of effective change management. This is discussed in the following sections. In this paper, we present an evolution-based configuration management model. In this model, the configuration management is divided into two domains, the *production domain* and the *product domain*. We suggest the evolution propagation of corrective changes and enhancement changes on different configuration artifacts follow different paths.

The remainder of the paper is organized as follows: Section 2 defines and clarifies the terminologies used in this paper. Section 3 reviews the current configuration management models in software product line and compares their advantages and drawbacks. We describe our evolution-based configuration management model in section 4. Section 5 contains the solutions to various configuration management issues in this model with emphasis on change management. The constraints and the conclusions are in Section 6.

2. Terminologies definition and clarification

To avoid the ambiguity and misunderstanding, we give the following definitions and explanations of the terminologies used in this paper:

Component: A component is the basic unit for configuration management. For example, a single file could be a component. A set of files that unite to perform a function or form an inheritance tree is also called a component. A component could be atomic or composite with respect to the composition of the internal files. However, in this paper, we treat both of them as the smallest configuration artifact.

Asset: An asset is a collection of components. The relation between component and asset is the part-whole relation. An asset may contain one or more components. There are two types of assets in a software product line, core asset and custom asset.

Core asset: A **c**ore asset contains a set of domain specific but application independent components that can be adapted and reused in various related applications. Core asset is one of the most important concepts in a software product line. In general, a core asset almost certainly includes an architecture that the products in the product line will share, as well as components that are developed for systematic reuse across the product line.

Custom asset: A custom asset contains a set of application specific components. A custom asset is not designed for reuse, but produced for a specific application. The quality requirement (for example, reusability) of custom assets is not as high as core assets and effort spent on maintenance of custom assets is less than on core assets.

Product: From a logical view, a product is a collection of core assets and custom assets. Software product line takes core assets and custom assets as input and produces a product as the output. Products share the same or similar core assets, because the input core assets need to be adapted to the specific product. A product can be logically considered to contain two parts, **core part** and **custom part,** which come from core assets and custom assets respectively. There are two types of products, *product instance* and *product in-use*.

Product instance: After a new product is produced, it may also need to be configuration managed. We call the product under configuration management *product instance*.

Product in-use: *Product in-use* is the product released to be used by the user. Therefore, in general, *product in-use* is a clone of a *product instance*.

Core instance: A *product instance* contains two parts, *core part* and *custom part*. The *core part* of a *product instance* is given the name *core instance*. Both the terminologies *core instance* and *product instance* are only used when we refer to configuration management.

Artifact: An artifact is a general term in software engineering. It refers to any manageable items produced in software development. Therefore, component, asset (core asset and custom asset), *core instance*, *product instance*, and *product in-use* are all artifacts.

Artifact evolution: Changes to an artifact may result in a new version of the artifact. We call this artifact evolution. There are two types of changes that can result in artifact evolution. They are corrective changes and enhancement changes.

Corrective change: *Corrective changes* are changes made to an artifact in order to remove a residual fault while leave the specification unchanged. It is also termed as "fixing change" or "repair." A corrective change is important

because, without it, the product cannot function the right way as specified in the functional requirement.

Enhancement change: Enhancement changes are changes made to an artifact in order to achieve a high quality non-functional requirement, such as security, performance, usability, and so on or to adapt to a new platform and a new functional requirement.

Evolution propagation: Software artifacts are interrelated. Changes to one artifact may require the corresponding changes to other artifacts. We call this evolution propagation. Usually, there are four basic paths for evolution propagation. They are *release path*, *request path*, *update path*, and *feedback path*. In this paper, we define a new path, *report path*.

Release path: If changes are made to a *product instance,* the product under configuration management, these changes need to be reflected in the *product in-use*. Thus, a new clone of *product instance* is released for the user. The evolution of the *product in-use* is affected by the evolution of *product instance*. We call this propagating direction of artifact evolution, the *release path*.

Request path: If change proposal to the *product in-use* comes from the user, the request should be first directed to the product under configuration management, the *product instance*. If the request is accepted, both the *product instance* and the *product in-use* will be modified. The evolution of the *product instance* is initiated by the *product in-use*. We call this propagating direction of artifact evolution, the *request path*.

Update path: If two artifacts under configuration management have part-whole relation, for example, artifact A1 is a part of artifact A2, changes to A1 may need to be reflected in A2. We call this propagating direction of artifact evolution, the *update path*. In a software product line, product is a composition of core assets and custom assets. Changes made to a core asset or custom asset may need to be reflected in the existing *product instance* that is previously produced from the original assets. The evolution of the *product instance* is affected by the changes to core assets or custom assets via update path.

Feedback path: If two artifacts (A1 and A2) under configuration management have part-whole relation, for example, A1 is a part of A2, changes to A2 may need to be reflected in A1. We call this propagating direction of artifact evolution, a *feedback path*. In a software product line, changes made to a *product instance* may also need to be reflected to the core asset and custom asset from which the product is built. The evolution of the core assets or custom assets is affected by the changes to the *product instance* via a feedback path.

Report path: If two artifacts (A1 and A2) are not tightly related, changes to A1 need not necessarily be reflected in A2. However, it may help A2's evolution if changes to A1 are known by A2. It is the decision of A2 whether to make changes accordingly and make the necessary changes. Hence the evolution of A1 is not reflected in A2 but informed to A2. We call this evolution propagation direction, the *report path*. The difference between *report path* and the other four paths is the propagation of changes (the other four paths) or the information (*report path*) about the changes.

3. Related models of configuration management in a software product line

Figure 1 shows the general configuration management model of a software product line, as proposed by [5] [14] [15] [17]. In this model, the artifacts under configuration management include core assets, custom assets, and *product instances*. Component is defined as the basic unit for configuration management. It should be included in any model. It is not shown in the figure. In this configuration management model, every *product in-use* has the corresponding *product instance* under configuration management. We use solid lines to represent the relationship between *product instance* and its clone *product in-use*. This is a one-to-one relation.

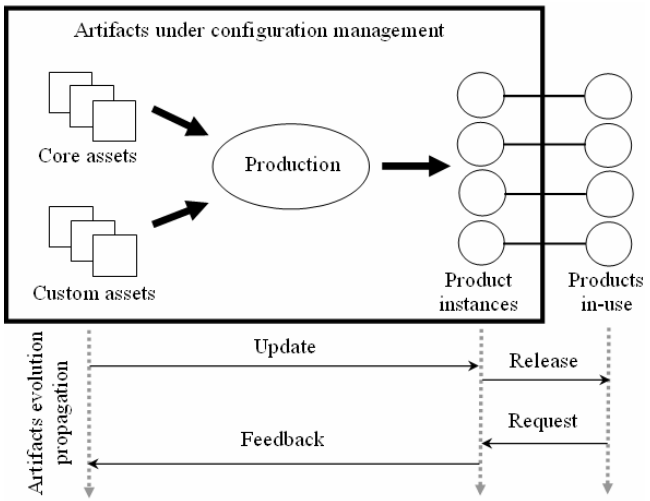

Figure1: General configuration management and asset evolution model for software product line [15]

The model in Figure 1 has several advantages: first, all artifacts in a software product line are under configuration management, which makes the changes, maintenance, and evolution of a software product line tightly controlled; second, this model allows for distributed software production, such as the separation of core assets development and product development. A new product can be built at the same time the assets are updated, because the product can be based on the existing version of assets; third, since every product is under configuration management, this

model allows the rapid reconstruction of any version of any product, which may have been built from various versions of core assets and custom assets.

However, this model also has some drawbacks as illustrated by Krueger [11]. First, consider the artifacts under configuration management. Each *product in-use* has the corresponding clone *product instance* being managed. For a specific product line, the number of products may be huge, which makes the task of configuration management too complicated.

Second, consider artifact evolution. As shown in Figure 1, in this model, there are four paths for artifacts evolution propagation. Changes made to core assets and custom assets cannot be directly applied to the product. They need to follow two consecutive paths, *update path* and *release path*: these changes need to be updated in *product instances*, followed by the release of new version the *product in-use*. On the other hand, changes made to *product in-use* cannot be directly reflected back to core assets and custom assets either. These changes need to be first requested to the corresponding *product instance*. If the changes are accepted, they will be made to both the *product in-use* and the *product instance* (with updated version of course). At the same time, these changes need to be fed back to the core assets and custom assets. A disadvantage of these scenarios is they do not differentiate the evolution of core assets and custom assets. In a software product line, the quality requirement and importance of core assets and custom assets are different. Hence, they should be managed differently. The fact that the evolution propagation of custom assets follows the same path as core assets increases the amount of configuration management work and complicates the evolution process.

Krueger [11] presented another configuration management model as shown in Figure 2, which he called *production line*. This model shows, only the core assets and custom assets are configuration artifacts (and components of course). Product is not under configuration management. He gave the following advantages of this model: (1) There is only one copy of core assets and custom assets to be configuration managed (the corresponding part in product is not under configuration management). This avoids the duplicate changes to product if changes are made to core assets or custom assets; (2) Changes made to core assets and custom assets during product development can be reflected in the product at the same time. (3) Because the product is not under configuration management, there is no necessity for evolution propagation between assets (core assets, custom assets) and the product.

Although the *production line* model has the above advantages, it has some drawbacks that make it dangerous to follow in practice. Consider the evolution of the configuration artifacts in the *production line* model shown in Figure 2, there is no direct dependency within artifacts under configuration management. Changes to core assets and custom assets cannot be updated to the *product in-use*, because the *product in-use* is not under configuration management. These changes are only available to future new products. Similarly, changes to *product in-use* cannot be fed back to the core assets and custom assets. The more changes to the *product in-use*, the more it will deviate from the product line. Therefore, the *production line* configuration management model ignores the most important property of a software product line, the interrelationships among different products. Without configuration management, it is difficult to track the origination of a product and reconstruct the product during maintenance.

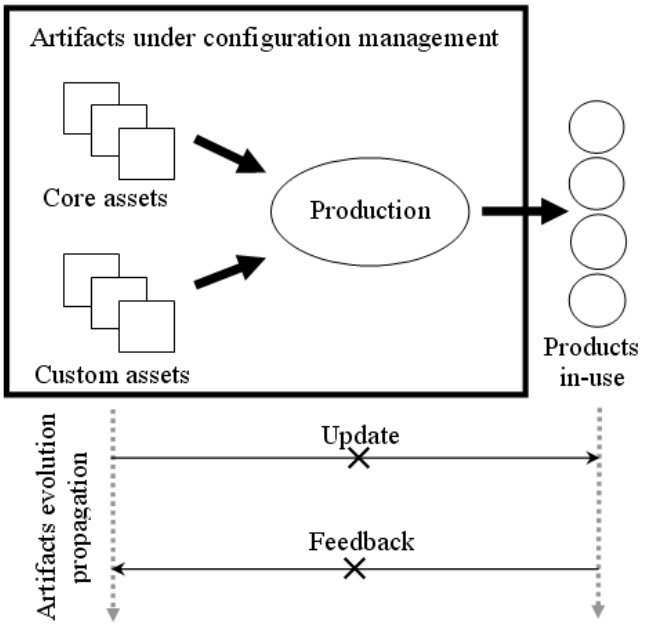

Figure2: *Production line* configuration management model proposed by Krueger [11]

Both the general model (Figure 1) and the *production line* model (Figure 2) treat different types of changes the same way. They do not differentiate *corrective changes* from *enhancement changes* on the artifacts. Consider the general model in Figure 1, if *corrective changes* are made to a product (*product instance* and *product in-use*), it is important that these changes be reflected back to the core assets or custom assets, because these changes may associate with a residual fault in core assets and custom assets. Therefore, it makes sense to change the corresponding assets and update all the related products in turn. However, if *enhancement changes* are made to a product (*product instance* and *product in-use*), it is not necessary to change the corresponding assets and update all other related products in turn. Because enhancement changes to one

product may not be appropriate for other products. On the other hand, if the corresponding assets are not changed according with the *enhancement changes* to a product, this product will deviate more from the product line, which may result in product line "decay" as described in Section 1.

4. Evolution-based configuration management model

In order to avoid the drawbacks of configuration management models presented in Section 2, we present an evolution-based configuration management model for a software product line, which is shown in Figure 3. In this model, the configuration management is divided into two domains, the *production domain* and the *product domain* (The word "domain" has been overloaded. It generally refers to an application area. Here, we use it to denote different configuration management processes). In *production domain*, the configuration artifacts are core assets, custom assets, and *core instances*. In *product domain*, the configuration artifacts are *product instances*. In this model, a software product line takes core assets and custom assets as input and produces *products instance* and *core instances* as output. The *core instances* are extracted directly from the corresponding *products instance*. We use dotted lines to represent the relation between *core instance* and *product instance*. Each *product instance* has one *core instance* under configuration management in *production domain*. Each *core instance* has one or more *product instances* that consist of it and is under configuration management in *product domain*. This relation is a one-to-many relation.

In the evolution-based configuration management model, every *product in-use* has a corresponding *core instance* under configuration management in *production domain*. As shown in Figure 3, changes to the core assets propagate to the *product in-use* via three consecutive paths, *update*, *update*, and *release*. Because core assets constitute the most important part of a product and the products differentiate mainly through core assets, it is vital that the changes made to core assets are available to the products and the changes to *core part* of a product are monitored by the configuration management in *production domain*. The evolution-based model provides this capability. Since both core assets and *core instances* are under configuration management in *production domain*, changes to core assets can be easily updated to *core instances*. If needed, the new version of *core instance* can then be updated to the corresponding *product instance* and released to the *product in-use*. On the other hand, change proposals to *core part* of a *product in-use* need to be first requested to the corresponding *product instance* followed by the same request being forwarded to *core instances*. If this proposal is accepted by the configuration management authority in *production domain*, the *core instance*, the *product instance*, and the *product in-use* will be changed. At the same time, the changes should be fed back to the core assets to make the corresponding changes.

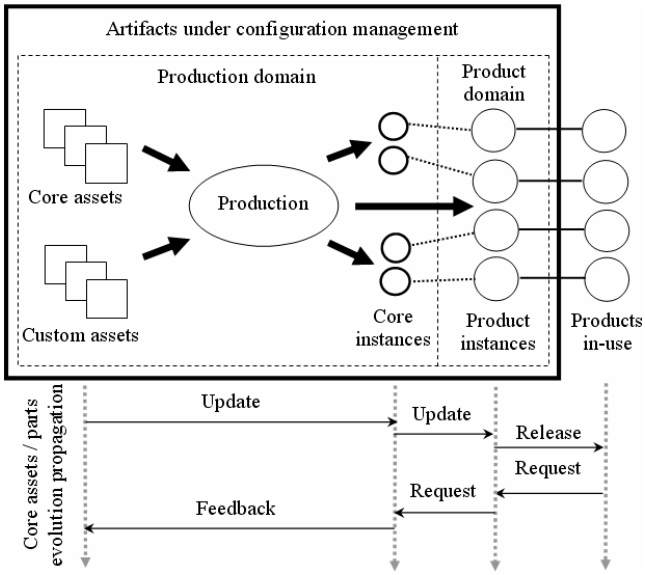

Figure 3: Evolution-based configuration management model for software product line with core assets and *core parts* evolution propagation

Figure 4 shows the evolution propagation of custom assets and *custom parts* in evolution-based configuration management model. Changes to custom assets need not to be updated in the corresponding *product instance* and *product in-use*. These changes are only available to future new products. Similarly, changes to the *custom part* of *product instance* and *product in-use* need not to be feedback to the custom assets, because the configuration management of *custom part* of a product is not connected with the configuration management of custom assets.

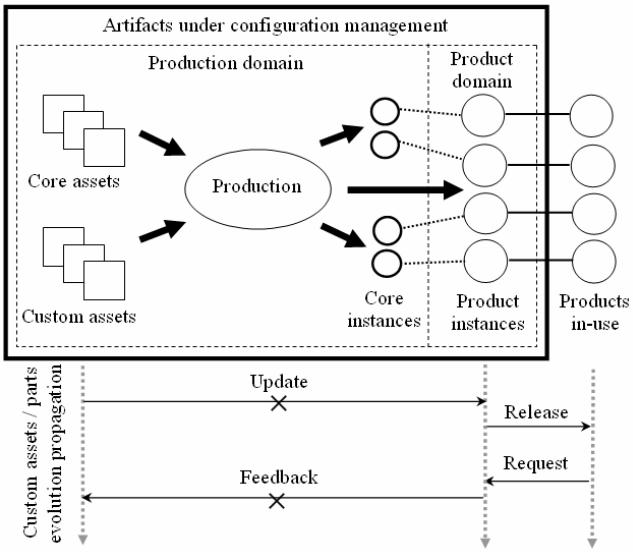

Figure 4: Evolution-based CM model for software product line with custom assets and *custom parts* evolution propagation

Comparing to the models presented in Figure 1 and Figure 2, the evolution-based configuration model has the following properties and advantages:

Clear separation of responsibilities: The configuration management of software product line is divided into two domains, the *production domain* and the *product domain*. The *production domain* manages the evolution of components, assets (core assets and custom assets), and *core instances*. The product domain manages the evolution of the product. However, the evolution of the *core part* of the product is monitored by the *production domain*. The *product domain* can only determine the evolution of *custom parts* of a product.

Multiple evolution propagation paths: We treat the evolution propagation of core assets and custom assets differently. The evolution of core assets and *core part* of a product is tightly connected, which is similar to the general model in Figure 1, while the evolution of custom assets and *custom part* of a product is not connected, which is similar to the *production line* model in Figure 2.

Mitigating product deviation: The evolution-based model can help prevent product line "decay." The changes to the *core part* of a *product instance* and *product in-use* are controlled by the configuration management in *production domain*. This can guarantee the changes will not result in product deviation from the product line. Changes to *custom part* of a *product in-use* are under separate configuration management in *product domain*. Because *custom parts* are not as important as *core parts*, changes to them usually cannot result in product line "decay."

5. Configuration management issues in the evolution-based model

In this section, we discuss the solution to various configuration management issues in the evolution-based model.

5.1. The artifacts and the associated configuration management operations

As we mentioned before, the basic configuration item is a component. Component can be configuration managed like conventional software. Operations for management of components are:

Version management: A version identifies a unique state of components as they evolve over time.

Branch management: A branch identifies a unique evolutionary direction of components. New branches can be created from the existing branch of the component. Different branches can merge into a single branch.

Concurrency management: Concurrency management is used to coordinate multiple modifications to the component by different developers. Usually, this is implemented using *checkout*, *checkin* policies.

Assets (core assets and custom assets) are collection of components. They can be treated as configurations or compositions of components. Operations for asset management are:

Asset creation: An asset is created by selecting a set of components.

Asset evolution: A change to the component in the asset will result a new version of asset.

Asset branch management: An asset could also have several evolutionary paths. Different paths can merge to a single path.

Asset baseline management: A baseline for an asset stands for a milestone of an asset. Determination of core asset baseline needs to be weighted carefully for their impact on the entire product line.

Core instance is a collection of core assets. They can be treated as configurations of assets. A product is a collection of core assets and custom assets. They are all under variant management:

Variant management: Coexisting core instances or products for different platforms are under variant management.

Different artifacts under configuration management will be managed using different operations. However, some basic operations are common to more than one type of artifacts. For example, version management is applied to all artifacts. Table 1 indicates the configuration management operation for various artifacts.

Table 1: Configuration management operations for various artifacts

Configuration management operations	Artifacts			
	Component	Asset	Core instance	Product instance
Version management	●	●	●	●
Concurrency management	●	●	●	●
Evolution management	●	●	●	●
Branch management	●	●		
Baseline management		●		
Variant management			●	●

5.2. Evolution management

The special property of the evolution-based model is the way it performs change management and coordinates the propagating of artifact evolution. In this section, we discuss evolution management in more detail.

An authorized group must carefully analyze any changes proposed to core assets and *core instance*, which includes the request from the user to make changes on *core instance* and the proposal from the developer to make changes on core assets. If the proposals are accepted, these changes could be reflected in and propagated to other artifacts under configuration management (either in *production domain* or in *product domain*).

We consider these changes differently on core assets, custom assets, and *product instances*. It should be noted here that there is no direct request of changes on *core instance*. *Core instance* is only part clone of the *product instance*. The request of changes to *core instance* comes from either change request to core assets or to the *product instance*. On the other hand, any request of changes to *product in-use* should be directed to the *product instance*. Therefore, we only consider the request of changes on *product instance*. We also consider two types of changes (corrective change and enhancement change) separately. Now we discuss the propagating of changes on assets:

Corrective changes to core assets: If corrective changes are made to core assets, these changes should be reflected in the corresponding *core instance*, the *product instance,* and the *product in-use*, because, it is important to fix a fault in a product. Therefore, the core assets, the *core instance*, and the product are updated to a new version. The corrective changes to core assets should propagate following the path *update*, *update*, and *release*.

Enhancement changes to core assets. If enhancement changes are made to core assets, these changes need not to be reflected in the corresponding *core instance* and the product. Because, there is no fault associated with the corresponding *product in-use*. Therefore, the changed core asset acquires a new version, while the *core instance*, and the product do not change according. They keep their current versions and are logically composed of the old version of the core asset. The enhancement changes to core assets do not need to propagate to existing product. They are only available to new products.

Corrective changes to *core part* of a product. If corrective changes are made to *core part* of a *product instance*, these changes should be reflected in the corresponding core asset and *core instance*. Again, this is because it is important to fix a fault in a product. The core assets, the *core instance*, and the product are updated to a new version. Therefore the corrective changes to *core part* of a *product in-use* should propagate following the path *request*, *request*, and *feedback*.

Enhancement changes to *core part* of a product. If enhancement changes are made to *core part* of a product, these changes should also be reflected in the corresponding core asset and *core instance*. Core assets are used to identify the origination of the product. To avoid product deviation from the product line, these changes must be monitored by the configuration management in production domain. Therefore the enhancement changes to *core part* of a *product in-use* should propagate following the path *request*, *request*, and *feedback*.

Corrective changes to custom assets. If corrective changes are made to custom assets, these changes are only available to new product. They need not be reflected in the existing products. So, corrective changes to custom assets do not need to propagate.

Enhancement changes to custom assets. If enhancement changes are made to custom assets, these changes need not to be reflected in the *product in-use* either. Hence, enhancement changes to custom assets do not need to propagate either.

Corrective changes to *custom part* of a product. If corrective changes are made to *custom part* of a product, these changes need not necessarily be reflected in the corresponding custom assets, since. custom assets are not intended to be reused in the future the same way as in the current product. However, corrective changes are associated with faults in an artifact. It will be helpful if these changes are noticeable by the configuration personnel in *production domain*. Therefore, these changes should be reported to custom assets. It is up to the configuration management personnel in *production domain* to decide if, and if so, what changes are to be made to the corresponding custom assets. Therefore, corrective changes to custom part of a *product in-use* needs to propagate following the path *request* (to the *product instance*), and *report* (to the custom asset).

Enhancement changes to *custom part* of a product. If enhancement changes are made to *custom part* of a product, these changes need not be reflected or reported to custom assets. Since the main difference is on *custom parts*, enhancement changes to *custom part* of a product do not need to propagate.

It should be noted that, changes (corrective change and enhancement change) to *custom part* of a product are performed and controlled in *product domain* only. All other changes need to be monitored by the configuration personnel in *production domain*. Table 2 summarizes the above discussions. The symbol "–" means no change propagating is necessary.

Table 2: Evolution propagating paths for various changes to artifacts in software product line

Type of changes	Artifact			
	Input asset		Product	
	Core asset	Custom asset	Core part	Custom part
Corrective	*update, update, release*	–	*request, request, feedback*	*request, report*
Enhancement	–	–	*request, request, feedback*	–

5.3. Versioning scheme

Versions of component, asset, and *core instance* can follow the conventional identification scheme. Versions of *product instance* should partially adhere to the product line. A version of a *product instance* should contain two parts. One part should enable it to be back track to the *core instance*. One possible approach is to use the same versioning scheme as the core instance. Another part is used to differentiate the continuous changes made to *custom part* of the product. This kind of versioning schemes can enable the developer to identify the origination of the product easily and reconstruct the product rapidly.

6. Constraints and conclusions

In this paper, we presented an evolution-based configuration management model. The special property of this model is the way it manages evolution propagation between related artifacts. We hope this model can contribute not only to software industry but also to manufacturing industry due to the similarities between them.

Despite several advantages mentioned in the earlier sections, the proposed model has the following constraints:

This model is for products in a product line that can easily extract *core parts* and *custom parts*. So, it is easy to differentiate core part and custom part and manage them differently. If core parts and custom parts mix in the product, it is difficult to manage them differently.

This model is best for products with the major part being the *core part*. Therefore, different products can share the same *core instance*. Management on *core instance* can reduce the working load of direct management on *product instance*. If core asset is a small part, it does not benefit too much from this model.

References

[1] Atkinson, C. *Component-Based Product Line Engineering with UML*, Addison-Wesley, 2001.

[2] Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.

[3] Bruegge, B. and Dutoit, A. H. *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Pearson Prentice Hall, Upper Saddle River, NJ, 2004.

[4] Burrows, C., George, G., and Dart, S. *Configuration Management*, Ovum Ltd, 1996.

[5] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[6] Dart S. Spectrum of functionality in configuration management systems, *Technical Report*, CMU/SEI-90-TR-11, Software Engineering Institute, Pittsburgh, PA, 1990.

[7] Dart, S. Concepts in configuration management systems, *Proceedings of the Third International Workshop on Software Configuration Management*, Trondheim, Norway, pp. 1–18, 1991.

[8] Estublier, J., Dami, S., and Amiour, M. High level process modeling for SCM systems, *Proceedings of the Seventh International Workshop on Software Configuration Management*, Boston, MA, pp. 81–97, 1997.

[9] Estublier, J. Software Configuration management: a roadmap, *Proceedings of the Conference on The Future of Software Engineering*, Limerick, Ireland, pp. 279–289, 2000.

[10] Feller P. Configuration management models in commercial environments, *Technical Report*, CMU/SEI-91-TR-7, Software Engineering Institute, Pittsburgh, PA, 1991.

[11] Krueger, W. Variation management for software production lines, *Proceedings of the Second International Conference on Software Product Lines*, San Diego, CA, pp. 37–48, 2002.

[12] Parnas, D. L. Software aging, *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, pp. 279–287, 1994.

[13] Perry, D. E., and Wolf, A. L. Foundations for the study of software architecture, *ACM Sigsoft Software Engineering Notes,* 17(4), 40–52, 1992.

[14] Software Engineering Institute, Software Product Lines, Carnegie Mellon University, http://www.sei.cmu.edu/productlines/index.html (accessed June 2005)

[15] Softwareproductlines.com, Software Product Lines, http://www.softwareproductlines.com/ (accessed June 2005)

[16] Staples, M. Change control for product line software engineering, *Proceedings of the 11th Asia-pacific Software Engineering Conference (APSEC'04)*, Busan, Korea, pp. 572–573, 2004.

[17] Weiss, D. and Lai, R. *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, Reading, MA, 1999.