

Development/Maintenance/Reuse: Software Evolution in Product Lines

Stephen R. Schach – Vanderbilt University, Nashville, TN, USA
Amir Tomer – RAFAEL, Haifa, Israel

Abstract

The evolution tree model is a two-dimensional model that describes how the versions of the artifacts of a software product evolve. The propagation graph is a data structure that can be used for effective control of the evolution of the artifacts of a software product. In this paper we extend the evolution tree model and propagation graph to handle the evolution of a software product line.

Software product lines are characterized by large-scale reuse, especially of core assets. We show how a third dimension can be added to the evolution tree model to handle this reuse. In particular, the new model incorporates bidirectional reuse within product lines. That is, the new model can handle the transfer of an artifact from the core assets repository to a specific product (acquiring a core asset) as well as the transfer of a specific asset from a specific product to the core assets repository (mining an existing asset).

1. Introduction

In classical software engineering, software construction consists of two activities performed sequentially: development and maintenance. Starting from scratch, the software product is developed and then installed on the client's computer. Any change to the software after installation, whether to fix a residual fault or extend the functionality, constitutes maintenance. Thus, the way that software was developed classically can be described as the *development-and-then-maintenance* model.

Nowadays, however, this model has proved to be impractical for two reasons. First, software engineers usually have to cope with changing requirements and evolving technologies, the so-called "moving target problem" [Schach, 1999]. To see how this affects the software life cycle, suppose that the client's requirements change while the design is being developed. The software engineering team will have to suspend development and modify the specification document to reflect the changed requirements. Furthermore, it may then be necessary to modify the design as well if the changes to the specifications necessitate corresponding changes to those portions of the design that have already been completed. Only when these changes have been made can development proceed. In other words, developers frequently have to perform software maintenance before the product is installed. In theory, such maintenance might have to be performed just one day after the client's original requirements were given to the software engineering team, in contrast to the classical model of development-and-then-maintenance.

Software reuse is the second reason why development-and-then-maintenance is no longer an accurate model for software construction. In classical software engineering, a characteristic of development is that the development team builds the target product starting from scratch. In

contrast, as a consequence of the high cost of software construction today, developers try to utilize existing software artifacts wherever possible. Reuse can be applied at different levels of abstraction. When a product is to include a graphical user interface (GUI), Microsoft Foundation Classes (MFC) are frequently reused. Moving to a higher level of abstraction, reuse of design patterns is becoming an increasingly popular way of reducing the design effort. Furthermore, when a design pattern is reused, the code that implements the pattern in question may also be reused. Taking this idea further, if a client's requirement for a target product has been implemented in an earlier product, then it should be possible to reuse the specification, design, test cases, documentation, code, and all the other artifacts associated with that requirement in the new product; this idea underlies software product lines.

Classically, the term "development" refers to constructing a new product from scratch. Thus, whenever an artifact is reused in the construction of a software product, it would be wrong to use the term "development" to describe the construction of that product, irrespective of the level of abstraction of the reused artifact. On the contrary, the presence of a reused artifact means that the software engineers start with an existing component that will need to be integrated into the target product. Thus, the development-and-then-maintenance model is inappropriate whenever there is any reuse. The prevalence of reuse is the second reason why the development-and-then-maintenance model is impractical these days.

More accurately, a software product is an evolving entity which starts from an initial implementation of a set of client's requirements and from then on undergoes frequent modifications, usually for purposes of increasing its functionality, appearance or performance, or in order to adapt to a changed environment. The evolution of a single product is modeled by the evolution-tree model [Tomer and Schach, 2000], briefly described in Section 2. The evolution process is expressed, in practice, by changes in requirements propagating through the software artifacts which implement these requirements, and which therefore need to be modified accordingly. The propagation graph model [Schach and Tomer, 2000], described briefly in Section 3, provides a practical method of dealing with controlled software modifications by means of a data structure and a repeated procedure.

A software product line is a "set of software-intensive products ... sharing a common, managed set of features ..." [Clements and Northrop, 1999]. These products usually reuse core assets, some of which originate in other products in the line. That is, the individual evolution of a product in the product line is affected by the evolution of other products, as well as the evolution of the set of core assets. The overall effect is the evolution of the entire product line, expressed by increasing its overall capabilities.

In this paper we extend the evolution-tree model to describe the evolution of a software product line. Then, we show how the propagation graphs of individual products interact, providing a practical and effective way to implement reuse.

In Section 2 we describe how the evolution of a product may be viewed by means of an evolution tree. In Section 3 the propagation graph, a data structure for controlling the changes to a software product, is described. The evolution of a software product line is discussed in Section 4, and software reuse within product lines is the topic of Section 5. In Section 6 we describe reuse of core assets. Conclusions and future work appear in Section 7.

2. Evolution of a Software Product

A software product is usually constructed through a phased process. The classical waterfall life-cycle model [Royce, 1970] is an example of such a process, where the transition from one

phase to the next depends on successful completion of the current phase. In the following we use a simple waterfall-like life-cycle model, comprising the following phases:

- *Requirements* – elicit the user's requirements
- *Analysis* – construct specifications based on the requirements
- *Design* – construct a design based on the specifications
- *Implementation* – implement an executable computer program based on the design.

Recent life-cycle models, such as the spiral model [Boehm, 1988] and the iterative/incremental model [Jacobson, Booch and Rumbaugh, 1999], view the development of a product as a set of iterations, resulting in a working version at the end of each iteration. However, within each cycle, a waterfall-like model is applied. When the development of a product is accomplished, it is usually delivered to the client and enters its maintenance phase. However, whenever maintenance is required the product once again goes through a waterfall-like process where its specification is usually modified to reflect the changed requirements, the design is modified to reflect the modified specification, and the implementation is modified to reflect the changed design. This modification process results in a new version (or release) of the product which is then delivered to the client. If no new requirements are imposed (as happens, for example, in corrective maintenance) the process of constructing a new version may start at a lower level in the waterfall, such as design or implementation. The product, therefore, evolves as new versions develop by means of the process described above. In the context of this paper we denote by *development* the waterfall-like process of constructing a new version of the product.

More closely, at each phase various software artifacts are constructed, at various levels of abstraction. These artifacts are reflected in the documents which indicate the completion of each phase. For example, using the terminology of UML [Rumbaugh, Jacobson and Booch, 1999], a requirements document may contain use cases, a specification document may contain class specification and class diagrams, and a design document may contain component diagrams and deployment diagrams. Each of these artifacts is subject to changes when new versions are developed. We denote here by *maintenance* the process of applying changes to the product's artifacts, following the definition of maintenance in ISO/IEC Standard 12207, which states that the maintenance process is activated when “software undergoes modifications to code and associated documentation due to a problem or the need for improvement or adaptation” [ISO/IEC, 1995].

Thus, we may view the evolution of a software product in two-dimensional space with axes:

- The development axis, along which versions are developed; and
- The maintenance axis, along which artifacts are modified

This description is the foundation of the evolution tree model, detailed in [Tomer and Schach, 2000]. Figure 1 depicts an example of an evolution tree. R_i , S_i , D_i , and C_i ($i = 0, \dots, 3$) denote versions of the requirements, the specification, the design and the code, respectively. The initial (empty) artifact, prior to any development action, is denoted by f (we assume that development is performed from scratch).

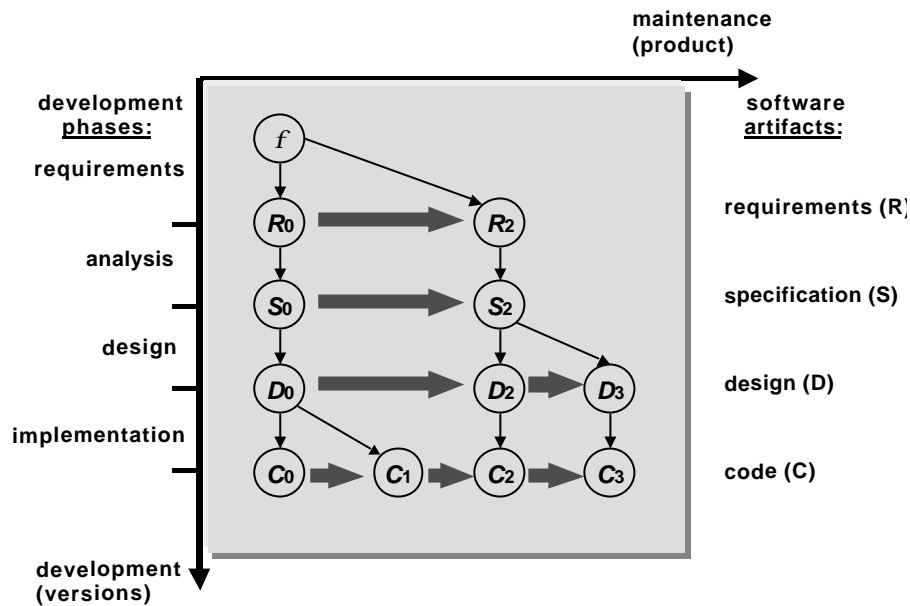


Figure 1: The two-dimensional evolution of a software product.

3. Software Maintenance with Propagation Graphs

The evolution tree model above describes the concept of software evolution. It also symbolically depicts the various versions of the requirements, specification, design, and code. A more detailed tool is required for controlling the changes in the product efficiently. One such tool is the propagation graph [Schach and Tomer, 2000], a data structure that can be used to control a maintenance-oriented software development process. The fundamental idea underlying a propagation graph is that software artifacts implement requirements and in turn generate new requirements. For example, a client's requirement is implemented by one or more specification artifacts. These specification artifacts in turn generate a new set of requirements, the specification requirements. This is illustrated in Figure 2, which shows the propagation graph for a specific case study described in [Schach and Tomer, 2000] using a simplified waterfall life-cycle model. The figure shows how software can be modeled as alternating sets of requirements and artifacts. Each set of artifacts satisfies the set of requirements above it and generates a new set of requirements below it.

For example, $ClientReq_1$ is implemented by three specification artifacts, $SpecArt_1$, $SpecArt_2$, and $SpecArt_4$. $SpecArt_1$ in turn generates specification requirement $SpecReq_1$, $SpecArt_2$ generates $SpecReq_2$, and $SpecArt_4$ generates $SpecReq_4$, $SpecReq_5$, and $SpecReq_6$. Next, $SpecReq_1$ is implemented in terms of $DesignArt_1$ and $DesignArt_4$, and so on.

More formally, a *propagation graph* G is a directed acyclic graph with $2n$ numbered sets of vertices. Vertices from odd-numbered sets are termed requirement vertices and vertices from even-numbered sets are termed artifact vertices. Arcs go from requirement vertices in set I to artifact vertices in set J with $I < J$ or from artifact vertices in set J to requirement vertices in set I with $J < I$.

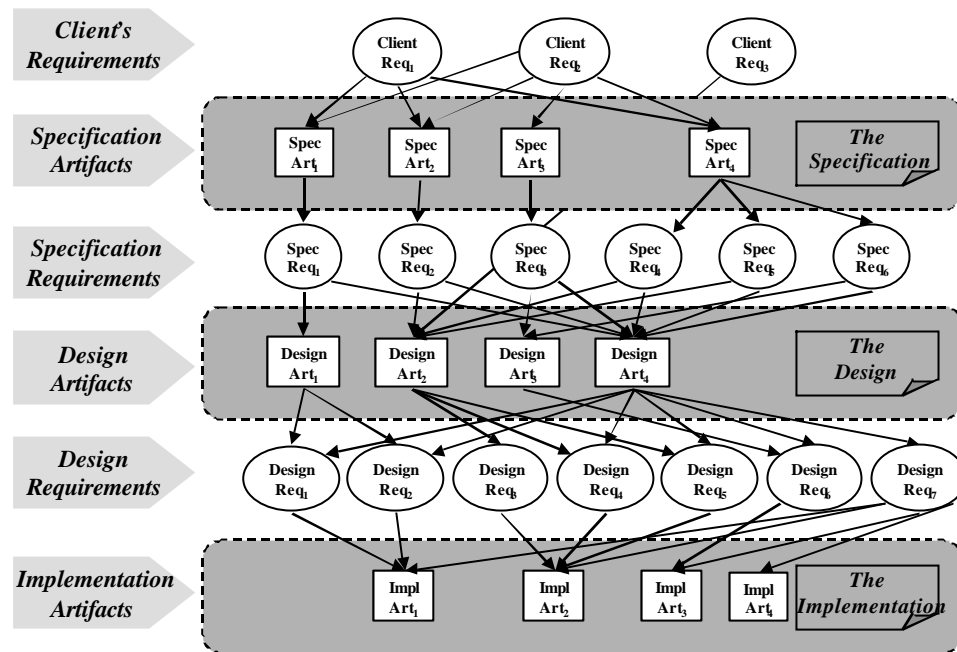


Figure 2. Propagation graph for a case study

As described in detail in [Schach and Tomer, 2000], the propagation graph is used to control software development and maintenance by means of a repeated stepwise procedure. At each step requirements are allocated (or reallocated) to artifacts. Then these artifacts implement the requirements, generating other requirements, as described above. These new requirements are then allocated (or reallocated) to other artifacts and so on. The efficiency of maintenance using propagation graphs is achieved by utilizing the allocation relation between requirements and artifacts. That is, when requirements change it is straightforward to locate the artifacts that are affected by these changes and therefore are expected to be modified. Other artifacts may safely be left unchanged.

Consider the propagation graph for a complete software product developed using an arbitrary software development methodology. The first set of vertices represents the client's requirements and the last set of vertices represents implementation artifacts. The remaining sets of vertices can be grouped into pairs; the first member of each pair represents a set of artifacts and the second member the requirements generated by those artifacts. For example, in Figure 2, one such pair is the specification artifacts together with the specification requirements generated by the specification artifacts.

In order to extend the propagation graph to product lines, we now define a *requirement-implementation subgraph (RISG)* of a requirement Req as the requirement together with all the artifacts and requirements that implement that requirement. More precisely, given a propagation graph G and vertex Req representing a requirement in that (directed acyclic) graph, the requirement-implementation subgraph of requirement Req, denoted by $RISG(Req)$, is the directed subgraph of G rooted at Req, that is, it consists of Req together with all its descendents in G .

For example, in Figure 2, the requirement-implementation subgraph of $ClientReq_1$ is the subgraph rooted at $ClientReq_1$. That is, $RISG(ClientReq_1)$ consists of the nodes $ClientReq_1$, $SpecArt_1$, $SpecArt_2$, $SpecArt_4$, $SpecReq_1$, $SpecReq_2$, $SpecReq_4$, $SpecReq_5$, $SpecReq_6$, together with all

four design artifact nodes, all seven design requirement nodes, and all four implementation artifact nodes.

In contrast, the requirement-implementation subgraph RISG (DesignReq₁) is the subgraph consisting of just two nodes, DesignReq₁ (the root) and ImplArt₁.

We will now extend the scope of Sections 2 and 3 from the individual product to software product lines. We first extend the evolution tree model, and then show how reuse is performed in practice using propagation graphs and requirement-implementation subgraphs.

4. Evolution of a Software Product Line

We now extend the evolution tree model of Section 2 to express the evolution of an entire software product line. A software product line is “a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission” [Clements and Northrop, 1999]. In practice, individual products are expected to evolve more efficiently within the product line framework, mainly by reusing software artifacts previously constructed during the evolution of other products in the product line. More specifically, a set of reusable artifacts, denoted as *core assets* [Clements and Northrop, 1999] is available for each of the products. Core assets may be obtained either by off-the-shelf acquisition from external sources (e.g., software libraries, design patterns), by development within the product line organization as generic building blocks, or by “mining” existing artifacts from specific products. Irrespective of their source, core assets are usually stored in a core assets repository and are maintained by a component-engineering group, external to any specific product development group. Products in a product line are derived from a common domain architecture, which is a core asset by itself.

Returning to the evolution tree, each of the individual products in the product line is represented by its own evolution tree. However, whenever a new artifact is to be constructed, it is likely that the core assets repository will be searched and reusable artifacts acquired for the product to be built. On the other hand, artifacts that have been constructed or maintained within an individual product may later be acquired back by the component-engineering group in order to be stored as new core assets in the repository.

We view a product line as a set of planes, each of which is associated with the two-dimensional evolution tree of a single product (see Figure 3). An extra plane (the top one in Figure 3) is dedicated to the core assets. Because the core assets, unlike specific products’ artifacts, may have been constructed, or acquired, independently of each other, the evolution tree of the core assets plane is not necessarily well structured. However, we are still able to classify these assets according to their level of abstraction (specification artifacts, design artifacts, code modules, etc.). Moreover, these artifacts relate to one another by means of allocated requirements, as will be detailed in the next section.

Thus, core assets are transferred from the core assets plane to specific products planes (acquiring a core asset), whereas specific artifacts may be transferred back to the core assets plane (mining an existing asset) in order to become core assets. This course of transfer may be viewed as a third dimension of the model, the reuse axis (see Figure 3). The overall evolution of the product line is expressed by the increase in the number of planes, associated with new products, as well as with the expansion of the core assets plane.

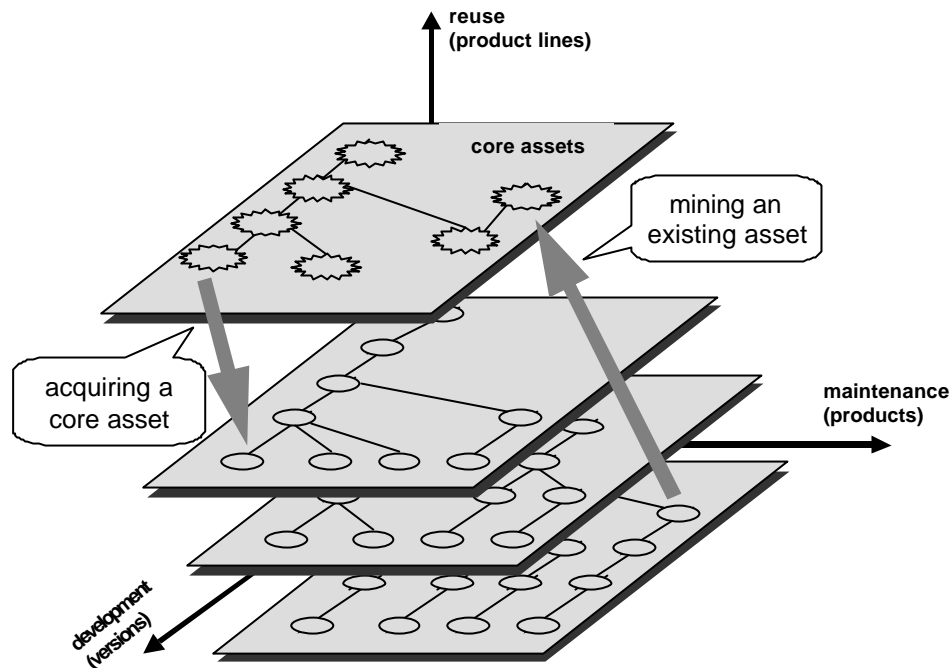


Figure 3: The three-dimensional evolution of a software product line

Although reuse may be handled independently of software product lines, the software product lines framework provides a more practical and effective context for software reuse [Boehm, 1999]. In the following we will show how propagation graphs, as described in Section 3 above, may be used for practical reuse applied to product lines.

5. Software Reuse in Product Lines

We now return to the propagation graph model and show how it can be applied to practical reuse. There are two major questions concerned with reuse: First, is reusing a core asset more effective than development of a new artifact (the development/acquisition decision [Clements and Northrop, 1999])? Second, if an artifact is of a higher level of abstraction (e.g., a design pattern), would it be effective to reuse it together with its descendents (e.g., a Java code module)? In the following we will address these two questions by means of the propagation graph model extended to a third dimension to incorporate the reuse inherent in product lines.

Suppose that a new product is to be constructed that will reuse existing software artifacts as much as possible. However, an artifact can be reused only if the requirements that it implements are also requirements of the new product. If there are such matching requirements, the new product can incorporate the requirement-implementation subgraph of those requirements.

Returning to Figure 2, suppose that ClientReq_1 is a client's requirement of a new product to be built. Then, if the artifacts of Figure 2 are to be reused, the new product will contain RISG (ClientReq_1). The problem with this, of course, is that RISG (ClientReq_1) implements far more than just client's requirement ClientReq_1 . In reality, RISG (ClientReq_1) includes all four implementation artifacts in Figure 2, and therefore implements all three client's requirements of Figure 2.

From this example it is clear that, when implementing requirement Req by reusing an existing implementation of that requirement, what is incorporated into the new product is not just RISG (Req) but rather the requirement-implementation subgraphs of *all* the requirements implemented by RISG (Req). We term these additional requirements *induced requirements*. In our example, there are two induced requirements, ClientReq₂ and ClientReq₃.

The presence of induced requirements in the new product may or may not pose a problem. One possibility is that the induced requirements implemented by RISG (Req) are also requirements of the new product. In this case, large-scale reuse can be achieved. Reuse on this scale has led to large cost savings [Matsumoto, 1987].

A second possibility is that the client's requirements for new product do not include one or more of the induced requirements. In this instance, reuse of RISG (Req) will result in a product that contains unnecessary artifacts. To take an extreme example, suppose that implementing ClientReq₁ could be achieved in only 300 lines of code, but RISG (ClientReq₁) is 500,000 lines long. Reuse of any superfluous material at all will inevitably lead to a maintenance nightmare because the maintenance team will have to determine precisely which portions of the product are superfluous but were included in the product in order to achieve reuse. Because of the maintenance implications, it is almost always wrong to reuse artifacts if this would result in induced requirements that are not also requirements of the target product.

A third possibility is that one of the induced requirements contradicts one or more of the client's requirements of the target product. In this case, no reuse is possible without drastic restructuring of the software; the cost of this restructuring may well make reuse impractical.

Similar problems can arise when reuse is employed at lower levels of abstraction as well. Suppose that the development team for the new product wishes to implement DesignReq₁ by reusing ImplArt₁ of Figure 2. As pointed out above, the requirement-implementation subgraph RISG (DesignReq₁) consists of just two nodes, DesignReq₁ (the root) and ImplArt₁. Nevertheless, problems can arise because ImplArt₁ also implements (all or part of) design requirements DesignReq₂ and DesignReq₇. Thus, there are induced requirements at the design requirements level, and the same reuse problems as with ClientReq₁ will arise here.

The problems of induced requirements cannot occur if each artifact is an implementation of just one requirement or, equivalently, if the propagation graph is a directed tree. In this situation, if the development team wishes to implement requirement Req, this is achieved by incorporating the requirement-implementation subtree of Req into the new product. This approach is applicable to accidental as well as planned reuse.

From the viewpoint of planned reuse, there is no problem. A characteristic of planned reuse is that the artifacts constructed for future reuse are independent entities. Thus, it is extremely unlikely that an artifact constructed as part of a program of planned reuse will have induced requirements.

On the other hand, based on our experience in the software industry, in general it is unusual for real-life software to consist of independent artifacts. Instead, the situation displayed in the case study (Figure 2) is typical of industry practice. The presence of induced requirements may be the reason why so many accidental reuse initiatives have been less than satisfactory.

However, there is one situation where accidental reuse is indeed enhanced by the presence of induced requirements, namely, software product lines.

In the software product line, each product at each plane has a propagation graph that describes its requirements-artifacts allocations. In particular, the core assets plane contains its own propagation graph, which maps requirements, common to the entire line, onto core artifacts. The

core assets propagation graph (CAPG) is a core asset by itself, because it can be reused whenever a new product needs to be derived from the domain architecture. In practice, this derivation is done by analyzing the requirements of the new product and deriving requirement-implementation subgraphs (RISGs), as appropriate.

Once the new product starts its own life cycle, its copies of reused core artifacts may be modified freely, according to specific requirements or specific design decisions. However, every such specific artifact is a candidate to become a new core asset (the “mining existing assets” practice). In order to do so, its specific relations with requirements over the specific product’s propagation graph should be analyzed and remapped into the CAPG. In other words, the relations between the CAPG and each specific propagation graph, as well as between any two propagation graphs in the product line, should be investigated in depth, with respect to common features of the product line, such as domain architecture and scoping.

6. Reuse of Core Assets

In the previous section we described how propagation graphs and their requirements-implementation subgraphs cater to reuse purposes. However, we used the term “reuse” merely for assets acquisition, that is, concentrating on the act of transferring an artifact from one plane to another. We now look more closely into various aspects of reuse.

There are two major categories of reuse [Schach, 1999]: *black-box reuse* in which an artifact is reused unchanged and *glass-box reuse* in which the artifact is modified in some way before it can be reused. Because we view reuse to be bi-directional (as shown in Figure 3), we need to refer to these reuse categories in both directions. Clearly, black-box reuse is associated both with core assets acquisition, when a copy of a core artifact becomes “as is” a specific product’s artifact, and with existing assets mining, when a copy of a specific product’s artifact becomes “as is” a core asset. In contrast, glass-box reuse is a compound action, involving both acquisition/mining and maintenance (as defined in Section 2). Let A be an artifact and A' be the same artifact after modification, denoted by $A' = \text{modify}(A)$. Now let another artifact B be an exact copy of A , denoted by $B = \text{copy}(A)$.

Let C denote a core asset and let P denote an artifact of a product. There are four possible cases of glass-box reuse:

- (a) $P = \text{modify}(\text{copy}(C))$. This is the case where a core asset is acquired by the product and is then modified to produce a specific artifact. It is a classical scenario of reusing a core asset as a baseline for a specific artifact. A typical example is where C is a design pattern and P is a specific design based on the pattern. It is likely that $\text{copy}(C)$ remains an asset of the specific product, for the purpose of documenting the design.
- (b) $P = \text{copy}(\text{modify}(C))$. This is the case where a core asset is modified before being acquired by the product. This scenario usually occurs when several products require a modified version of the core asset at the same time. A typical example is where the component-engineering group develops a specific design pattern and then copies of that pattern are released for acquisition. Here, too, it is likely that both C and $\text{modify}(C)$ will be retained as core assets.
- (c) $C = \text{modify}(\text{copy}(P))$. This is the case where a specific product’s artifact is mined but needs to undergo modifications before becoming a core asset. It is a typical scenario when establishing an initial core assets repository from specific existing products, consistent with a standard architecture. For example, suppose that CORBA has been adopted as a standard for object sharing. Then each specific class needs to be “wrapped”

with IDL (Interface Definition Language) elements before being qualified as a reusable core asset.

- (d) C = copy (modify (P)). This is the case where a specific product's artifact undergoes modifications before being mined as a core asset. This is the least likely scenario for reuse, because the core assets plane is not expected to "record" modifications performed within specific products. However, this could be the case when the component-engineering group notices that the product has upgraded an artifact which was previously mined as a core asset. In this case they would copy the modified artifact as a modified version of the core asset.

In terms of our three-dimensional evolution model, the copy and modify operations may be described as "moves" of artifacts along the axes, where copy refers to moves along the reuse axis between planes, and modify refers to moves along the maintenance axis within the same plane.

(For completeness, the operation of the third dimension, along the development axis, may be defined as implement, where $B = \text{implement}(A)$ denotes that B is an implementation of A.

7. Conclusions and Future Work

In this paper we introduced a three-dimensional model that describes the evolution of software product lines and outlined the means by which propagation graphs may be employed for effective reuse purposes within the context of product lines.

The characteristics of propagation graphs and their associated requirement-implementation subgraphs should be studied in detail and practical methods for using them should be developed.

We are currently building CASE tools to support the evolution tree and propagation graph models. In this paper we have added a third dimension to the evolution tree to describe the reuse inherent in software product lines. It will be necessary to extend those CASE tools to support this new development.

8. References

- [Boehm, 1988] B. BOEHM, "A Spiral Model of Software Development and Enhancement," *IEEE Computer* 21 (May 1988), pp. 61-72.
- [Boehm, 1999] B. BOEHM, "Managing Software Productivity and Reuse," *IEEE Computer* 32 (September 1999), pp. 111-113.
- [Clements and Northrop, 1999] P. CLEMENTS AND L. M. NORTHROP, *A Framework for Software Product Line Practice - Version 2.0*, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, July, 1999.
- [ISO/IEC, 1995] "Software Life Cycle Processes," ISO/IEC 12207, International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.
- [Jacobson, Booch and Rumbaugh, 1999] I. JACOBSON, G. BOOCH, AND J. RUMBAUGH, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.
- [Matsumoto, 1987] Y. MATSUMOTO, "A Software Factory: An Overall Approach to Software Production," in: Tutorial: Software Reusability, P. Freeman (Editor), Computer Society Press, Washington, DC, 1987, pp. 155-78.
- [Rumbaugh, Jacobson and Booch, 1999] J. RUMBAUGH, I. JACOBSON AND G. BOOCH, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA, 1999.
- [Royce, 1970] W. W. ROYCE, "Managing the Development of Large Software Systems: Concepts and Techniques," *1970 WESCON Technical Papers, Western Electronic Show and*

- Convention*, Los Angeles, August 1970, pp. A/1-1–A/1-9. Reprinted in: *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, May 1989, pp. 328–38.
- [Schach, 1999] S. R. SCHACH, *Classical and Object-Oriented Software Engineering with UML and C++*, 4th Edition, McGraw-Hill, 1999.
- [Schach and Tomer, 2000] S. R. SCHACH AND A. TOMER, “A Maintenance-Oriented Approach to Software Construction,” *Journal of Software Maintenance—Research and Practice* **12** (January/February 2000) (to appear).
- [Tomer and Schach, 2000] A. TOMER AND S. R. SCHACH, “The Evolution Tree: A Maintenance-Oriented Software Development Model,” *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR 2000)*, Zurich, Switzerland, February/March 2000 (to appear).