

## A Software Product Model Emphasizing Relationships

SHIH-CHIEN CHOU AND CHUN-WEI HUANG

*Department of Computer Science and Information Engineering  
National Dong Hwa University  
Hualien, 974 Taiwan  
E-mail: sczhou@mail.ndhu.edu.tw*

Managing software products during software development is essential. A model is needed to model products and facilitates product management. Important product management functions include: 1) version control, 2) configuration management, 3) product consistency management, 4) reference completeness management, and 5) product reuse support. This paper proposes a product model to facilitate the functions above. Since product relationships play important roles in product management, the proposed model especially emphasizes them.

**Keywords:** product management, product relationship, product model, version control, configuration management

### 1. INTRODUCTION

During software development, various products (software products, including documents and program code) will be produced. Product management is thus essential. In managing products, a model is needed. In addition to products, product relationships should also be modeled. Perhaps the most general relationship to model is the version relationship, with which product versions can be controlled and product configuration can be managed.

Version relationship, however, is not the only relationship to model. Various inter- and intra-product dependency relationships should also be modeled. *Inter-product dependency relationships (inter-DEP relationships)* exist among products or sub-products of different products. For example, if a design document is developed based on (i.e., referring to) a specification, an inter-DEP relationship exists between the two products. On the other hand, *intra-product dependency relationships (intra-DEP relationships)* exist among sub-products of the same product. For example, if a module of a program invokes another one, an invocation relationship (which is an intra-DEP relationship) exists between the two modules. Generally, inter- and intra-DEP relationships are established according to *change effects*. For example, if a specification is changed, the design document(s) inter-dependending on the specification will be affected and should be changed accordingly.

The use of version relationship such as version control and configuration management is well-known. Below we describe what can be benefited from intra- and inter-DEP relationships.

---

Received March 8, 2002; revised March 13 & June 23, 2003; accepted September 8, 2003.  
Communicated by Michael R. Lyu.

- 1) Intra-DEP relationships. Intra-DEP relationships link sub-products within a product. To construct intra-DEP relationships within a product, the product should first be decomposed into sub-products. Intra-DEP relationships are then established among the sub-products according to change effects. When a sub-product of a product is changed, other sub-products affected by the change can be identified by tracing intra-DEP relationships. We use an example to explain in more details what can be benefited from intra-DEP relationships. Suppose we have a program of 10K lines and the program is stored in a repository as a single product (i.e., the program is not decomposed and no intra-DEP relationship is established). Then, when a portion of the program is changed, the whole program should be checked to identify the parts that are affected by the change. Since the size of the program (i.e., 10K lines) is large, identifying the parts affected by a change within the program would be time-consuming and error-prone. On the other hand, suppose the 10K-lined program is decomposed into 20 sub-products and intra-DEP relationships are established among the sub-products. Then, when a sub-product is changed, intra-DEP relationships can be traced to identify the affected sub-products. The affected sub-products can then be checked to identify the affected parts. Since the size of a sub-product (i.e., 0.5K lines in average) is much smaller than that of the whole product, identifying affected parts from sub-products should be much easier than identifying affected parts from the whole product. Since product change is unavoidable during software development and maintenance, intra-DEP relationships contribute much in both software development and maintenance.
- 2) Inter-DEP relationships. Inter-DEP relationships are actually the “developed based on” relationships. For example, if a design document is developed based on a specification, the former product inter-dependes on the latter one. Since changing a product will affect the ones inter-depending on the changed product, inter-DEP relationships facilitate identifying the affected products when a product is change. Combining product decomposition and inter-DEP relationships contribute more in identifying the exact parts within the affected products. That is, if inter-DEP relationships are established between sub-products of different products, identifying the affected parts within an affected product will be much easier. We use an example to explain this. Suppose a design document is developed based on a specification and both the products are not decomposed. In this case, an inter-DEP relationship exists between the products. When a portion of the specification is changed, the whole design document should be checked to identify the affected parts. On the other hand, suppose the specification and the design document are decomposed and inter-DEP relationships are established between the sub-specifications and sub-design documents. Then, when a sub-specification is changed, only the sub-design document(s) inter-depending on the changed sub-specification should be checked to identify the affected parts. Since the size of a sub-product is smaller than that of the whole product, identifying affected parts from a sub-product will be easier than from the whole product.

By now we only describe the use of product relationships in change management. In our research, we also identified that product relationships facilitate identifying *correlated products* for reuse, as described below:

- 1) Products developed based on the reused one may be reusable. For example, when a specification is reused, the design document developed based on it may be reusable. Here the design document is *correlated to* the specification.
- 2) When a sub-product is reused, other sub-products of the same product may be reusable. For example, when the book management subsystem of a library system is reused, the borrower management subsystem may be reusable. Here the sub-systems are *mutually correlated*.
- 3) When a product is a candidate for reuse, revisions of the product can be considered alternative selections for reuse. Here revisions are mutually correlated.

As a summary, product relationships play an important role in product management. We thus propose that product relationships should be emphasized in a product model. Important product management functions and the use of product relationships in the functions are described below:

- 1) Controlling and merging product versions. Version control is essential in product management. Version merge prevents keeping too many versions. This function needs the support of version relationships.
- 2) Managing product configurations. According to versions, a product may have multiple configurations. When a product is accessed, the right configuration should be retrieved. This function needs the support of version relationships.
- 3) Managing product consistency (managing product change). Changing a product may affect others, which may in turn affect still others. This results in *inter-product change ripple effects*. Moreover, changing a sub-product of a product may affect other sub-products of the same product, which may in turn affect still others. This results in *intra-product change ripple effects*. Both effects should be handled to manage product consistency [1]. Since the products affected by a change can be identified by tracing product relationships [2], inter- and intra-DEP relationships respectively facilitate managing inter- and intra-product change ripple effects.
- 4) Managing reference completeness. We use an example to explain reference incompleteness. Suppose that a design document depends on a specification. If a new version has been created for the specification, then, ideally, a new version of the design document should be developed based on the new specification version. If no such design document is developed, the new specification version is not referred to. We call this situation reference incompleteness. Reference incompleteness indicates that some products are undeveloped. This function needs the support of both version and inter-DEP relationships.
- 5) Product reuse support. Product reuse (i.e., software reuse) has been recognized as important [3]. Many product reuse techniques have been developed, such as that for classifying and retrieving software components [4], that for developing reusable components [5], and so on. We are not proposing a product reuse technique. We, instead, emphasize that product relationships facilitate identifying correlated products for reuse.

We surveyed quite a few models for configuration management and version control [6-24] (see section 5 for the discussion of related work), and identified that some of them

manage configurations and versions within a single product [6, 8, 10, 13-14, 17-21, 24]. With these models, product management across products such as inter-product change ripple effects management, reference incompleteness management, and identification of correlated products for reuse cannot be achieved. Although we have identified models that manage inter-DEP relationships [11, 15], they manage poor intra-DEP relationships. This causes difficulty in intra-product management such as intra-product change ripple effects handling. Since we cannot identify a model that manages product relationships well, we developed a new model. This article proposes the model and describes product management using the model.

## 2. THE MODEL

The model models products and product relationships. The following subsections explain the model using examples and give formal definitions.

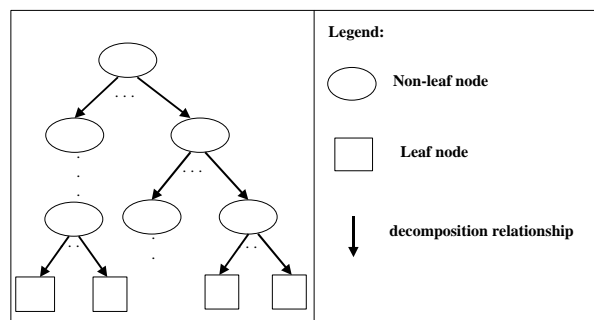


Fig. 1. Product hierarchy.

### 2.1 Product

A product can be decomposed into several sub-products, which can be further decomposed if necessary. Decomposing a product thus results in a product hierarchy as shown in Fig. 1. The entire hierarchy in the figure is a product. A leaf node or a sub-hierarchy rooted at a non-leaf node other than the root is a sub-product. Only leaves in a product hierarchy are real documents. Non-leaves are used to structure their children. Fig. 2 depicts a partial product hierarchy for a library management system written in the C language. The program template is shown in Fig. 2 (a). The corresponding product hierarchy is depicted in Fig. 2 (b), which shows that the system is composed of the modules “main” and “borrowerMng”, and the subsystem “bookMngSubsyst”. Moreover, the subsystem “bookMngSubsyst” is composed of the modules “bookMng”, “borrowBook”, and “returnBook”. Fig. 3 depicts a partial product hierarchy of a supermarket system’s specification represented in DFDs. Figs. 3 (a), (b), and (c) respectively show the context diagram, diagram 0 DFD, and diagram 1 DFD. Fig. 3 (d) is the corresponding product hierarchy, which shows that the system is composed of the “order management subsystem” and the following documents: context diagram, diagram 0 DFD, and the mini-spec

of “stock management”. Moreover, the order management subsystem is composed of the following documents: diagram 1 DFD, and the mini-specs of “in-order management” and “out-order management”.

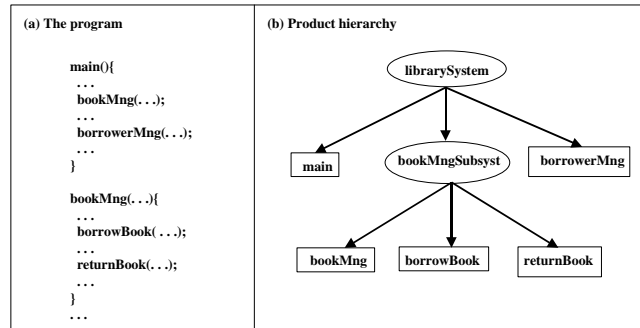


Fig. 2. Partial product hierarchy for a C program.

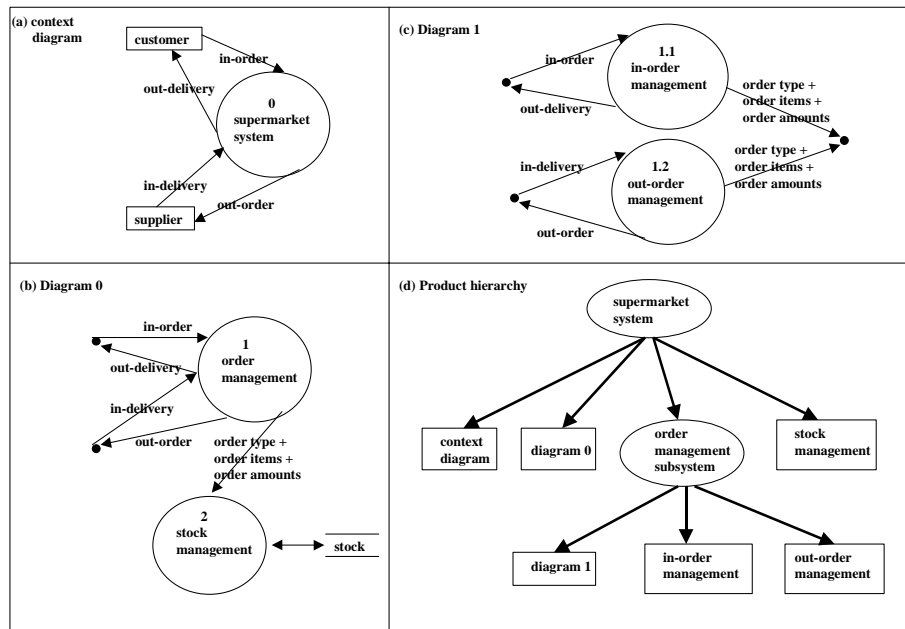


Fig. 3. Partial product hierarchy for a specification.

In addition to decomposition relationships, other intra-product relationships may be established according to change effects. For example (see Fig. 2), an invocation relationship exists between the module “main” and “bookMng”, in which the former invokes the latter. In this case, changing “bookMng” may affect “main”. As another example (see Fig. 3), a dependency relationship exists between the “order management subsystem”

and diagram 1 DFD (i.e., changing the latter may affect the former because the later outlines the former), between the documents “stock management” and “in-order management” (i.e., changing the latter may affect the former because the former accepts data from the latter), and so on. We define intra-product relationships other than product decompositions as intra-DEP (inter-product dependency) relationships. As will be described later, intra-DEP relationships are useful in handling intra-product change ripple effects. Figs. 4 (a) and (b) respectively depict the revised product hierarchy of Fig. 2 (b) and Fig. 3 (d) with intra-DEP relationships added.

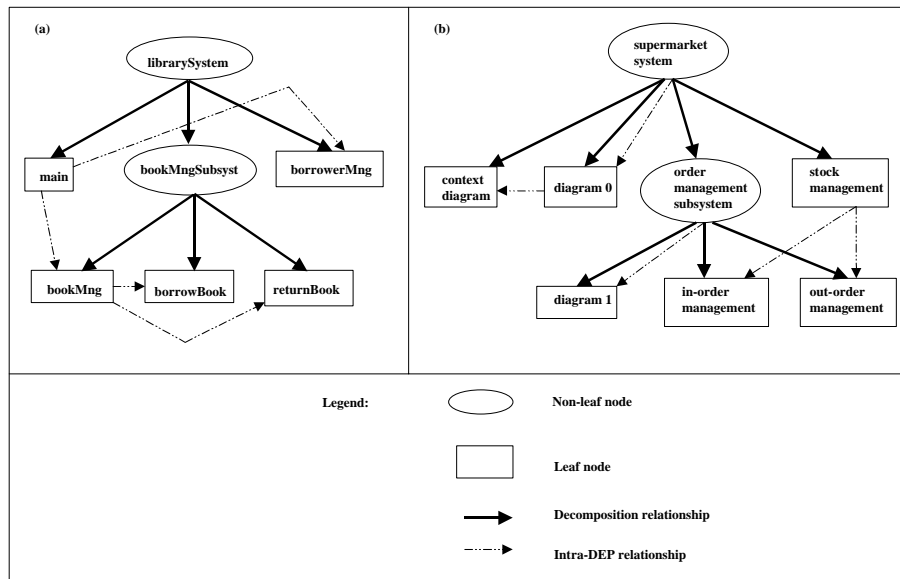


Fig. 4. Product hierarchy with intra-DEP relationships.

A product or sub-product may have multiple versions. The proposed model uses a tree similar to that in RCS [7] to structure versions. We call the tree a *version tree*. To number the nodes in a version tree, the first version is numbered as  $vI$ . Revisions of  $vI$  are respectively numbered as  $vI.1$ ,  $vI.2$ , and so on. Since our model does not differentiate revisions from variants, the words “revision” and “new version” are used interchangeably.

To model versions, each node in a product is extended to be a version tree. If a node has only one version, its version tree is composed of one node. Every version tree has a *default version*, which is generally the latest created one. Fig. 5 sketches a product hierarchy extended by version trees for Fig. 4 (a), in which two kind of version trees are presented, namely version trees of leaves and those of non-leaves. Fig. 5 is called a *product*. We use directed graphs to define a product and the terms related to a product, in which nodes and relationships in a product respectively correspond to vertexes and edges in a directed graph. We first define a version tree of leaves ( $G_{vll}$ ), that of non-leaves ( $G_{vml}$ ), and version relationship ( $E_{r-version}$ ) as follows:

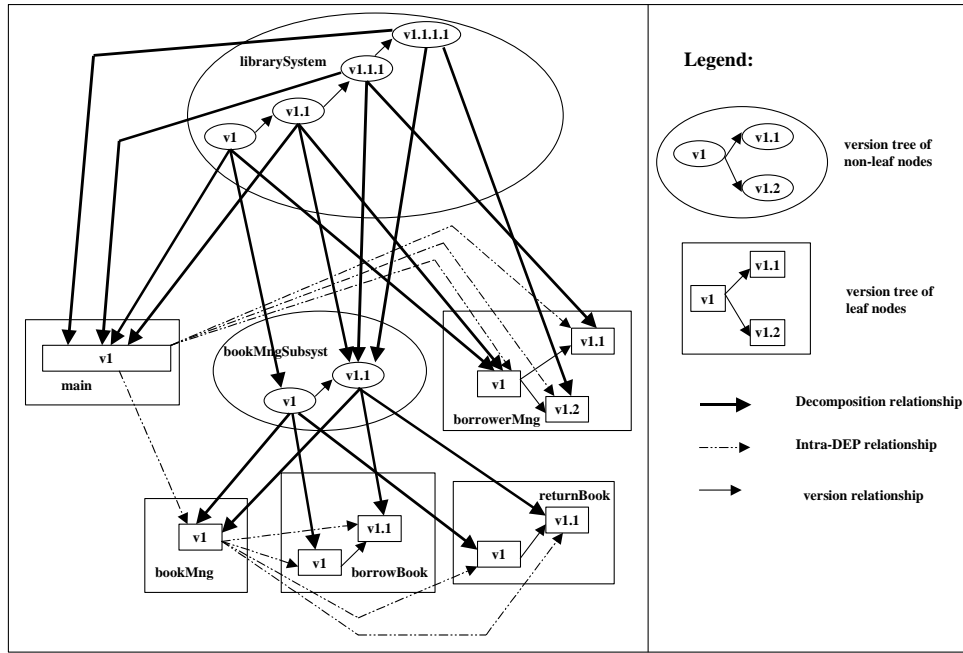


Fig. 5. Product (product hierarchy extended with version trees).

### Definition 1

$$G_{vnl} = (V_{vnl}, E_{r-version})$$

$$G_{vnl} = (V_{vnl}, E_{r-version})$$

$$V_{vnl} = \{v \mid v \text{ is a node in a version tree of leaves}\}$$

$$V_{vnl} = \{v \mid v \text{ is a node in a version tree of non-leaves}\}$$

$$E_{r-version} = \{\langle u, v \rangle \mid u, v \in V_{vnl} \cup V_{vnl} \wedge v \text{ is a revision of } u\}$$

In Definition 1, a node in a version tree of leaves may be program module(s) (see Fig. 2) or a sub-document (see Fig. 3). Moreover, a node in a version tree of non-leaves structures its sons using decomposition relationships.

Based on Definition 1, we define a product ( $G_{pd}$ ), the decomposition relationships ( $E_{r-decomposition}$ ), and the intra-DEP relationships ( $E_{r-intraDEP}$ ) as follows:

### Definition 2

$$G_{pd} = (V_{vt}, E_{rel})$$

$V_{vt}$  is a vertex set, in which each vertex corresponds to a version tree.

$$E_{rel} = E_{r-decomposition} \cup E_{r-intraDEP}$$

$$E_{r-decomposition} = \{\langle u, v \rangle \mid u \in V_{vnl} \wedge v \in V_{vnl} \cup V_{vnl} \wedge v \text{ is a sub-product of } u\}$$

$$E_{r-intraDEP} = \{\langle u, v \rangle \mid u, v \in V_{vnl} \cup V_{vnl} \wedge u \text{ intra-depends on } v\}$$

Definition 2 defines two types of relationships, namely decomposition and intra-DEP relationships. They link sub-products within the same product.

According to versions, a product may have multiple *configurations*. A configuration is a version of product or sub-product. It is composed of nodes structured into a tree by decomposition relationships. Moreover, intra-DEP relationships between the nodes are also presented. Fig. 6 shows some configurations of the product in Fig. 5. Based on Definitions 1 and 2, we define a configuration ( $G_{cf}$ ) as follows.

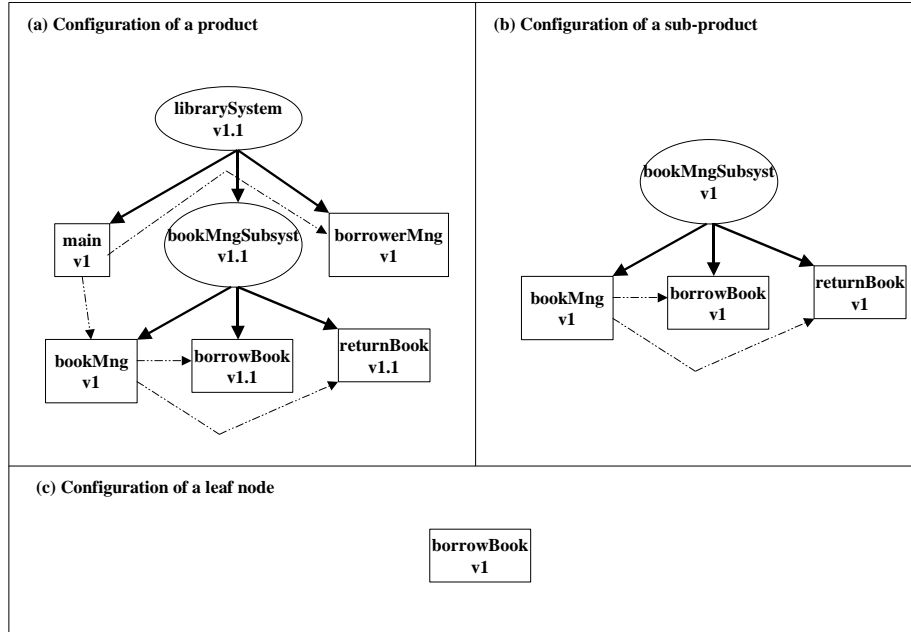


Fig. 6. Some configurations of the product in Fig. 5.

### Definition 3

$$G_{cf} = (V_{cf}, E_{cf})$$

$V_{cf}$  is a vertex set, which is composed of  $G_{cf}$ 's root and all descendants of the root

$$E_{cf} = \{ \langle u, v \rangle \mid u, v \in V_{cf} \wedge \langle u, v \rangle \in E_{r-decomposition} \cup E_{r-intraDEP} \}$$

## 2.2 Product Space

During software development, multiple products may be developed, in which each product can be represented by a  $G_{pd}$  defined in Definition 2. Inter-product dependency (inter-DEP) relationship may exist among products (or sub-products of different products). For example, if a design document is developed based on a specification, the design document inter-dependes on the specification. Fig. 7 shows inter-DEP relationships between a specification and its corresponding design document. Note that the decomposition granularity of two products may be different even if inter-DEP relationships exist between them.

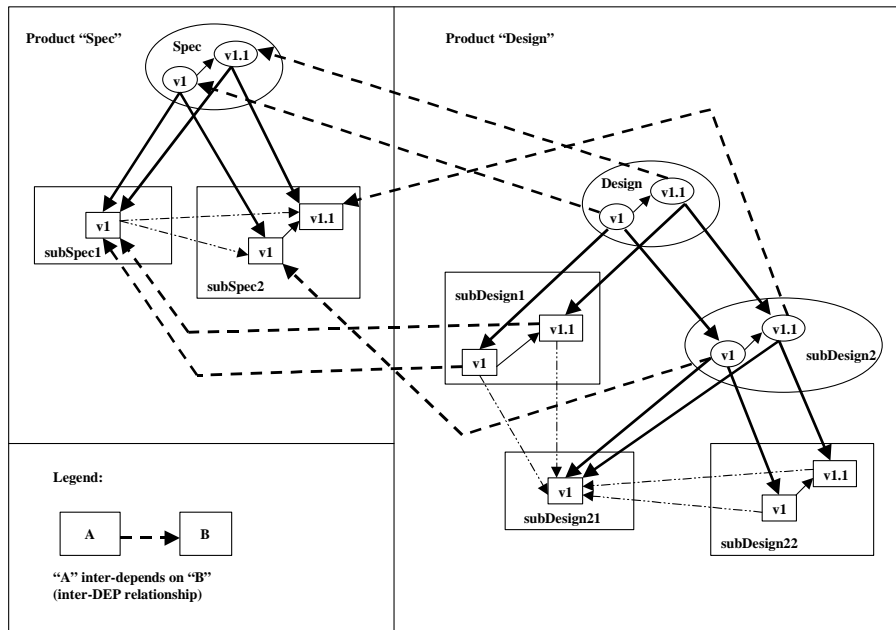


Fig. 7. Inter-DEP relationships.

We use a *product space* to model products and their inter-DEP relationships. Based on Definitions 1 and 2, we define a product space ( $G_{ps}$ ) and inter-DFP relationships ( $E_{r-interDEP}$ ) as follows:

#### Definition 4

$$G_{ps} = (V_{pd}, E_{r-interDEP})$$

$V_{pd}$  is a vertex set, in which each vertex corresponds to a product defined in Definition 2

$$E_{r-interDEP} = \{ \langle u, v \rangle \mid u, v \in V_{vfl} \cup V_{vml} \wedge u \text{ inter-depends on } v \}$$

Definition 4 defines product space and inter-DEP relationship. Inter-DEP relationships link sub-products of different products. Sub-products linked by inter-DEP relationships belong to different levels of abstraction. For example, inter-DEP relationships exist between sub-product of a specification and those of a design document, in which the specification and the design document are in different levels of abstraction. Note that inter-DEP relationship is a totally different concept when comparing with decomposition and intra-DEP relationships (see Definition 2), because the latter two types of relationships link sub-products within the same product.

### 3. PRODUCT MANAGEMENT

In this section, we describe product management functions using the proposed model, including version control and merge, configuration management, product consistency management, reference incompleteness management, and product reuse support.

### 3.1 Version Control and Merge

A configuration can be checked out to create a new version. When the new version is checked back in, the version control process is invoked. We first explain the process using examples then conclude it with an algorithm. To simplify the discussion, we call the checked out configuration  $G_{cf\_ori}$ , the new version  $G_{cf\_nver}$ , and the product containing  $G_{cf\_ori}$   $G_{pd1}$ .

Suppose that the configuration in Fig. 8 (b) is checked out from the product in Figure 8 (a) to create the new version in Fig. 8 (c). Here, Figs. 8 (a), (b), and (c) respectively correspond to  $G_{pd1}$ ,  $G_{cf\_ori}$ , and  $G_{cf\_nver}$ . The symbol “(\*)” in Fig. 8 (c) indicates that “LD v1” and “LC v1.1” get a new version. The following node types may be included in  $G_{cf\_nver}$ :

- 1) *New nodes*, which are nodes in  $G_{cf\_nver}$  but not in  $G_{cf\_ori}$ . For example, the node “LZ” in Fig. 8 (c) is a new node.
- 2) *RevisionNode*, which may be: (a) a leaf node that gets a new version or (b) a non-leaf node with one or more children that are revisionNodes or new nodes. Example revisionNodes in Fig. 8 (c) are “LD v1” and “NLC v1”.
- 3) *Unchanged nodes*, which are neither revisionNodes nor new nodes.

Sometimes, there may exist nodes that are in  $G_{cf\_ori}$  but not in  $G_{cf\_nver}$ . They are called *deleted nodes*. For example, “LE v1” in Fig. 8 (b) is a deleted node.

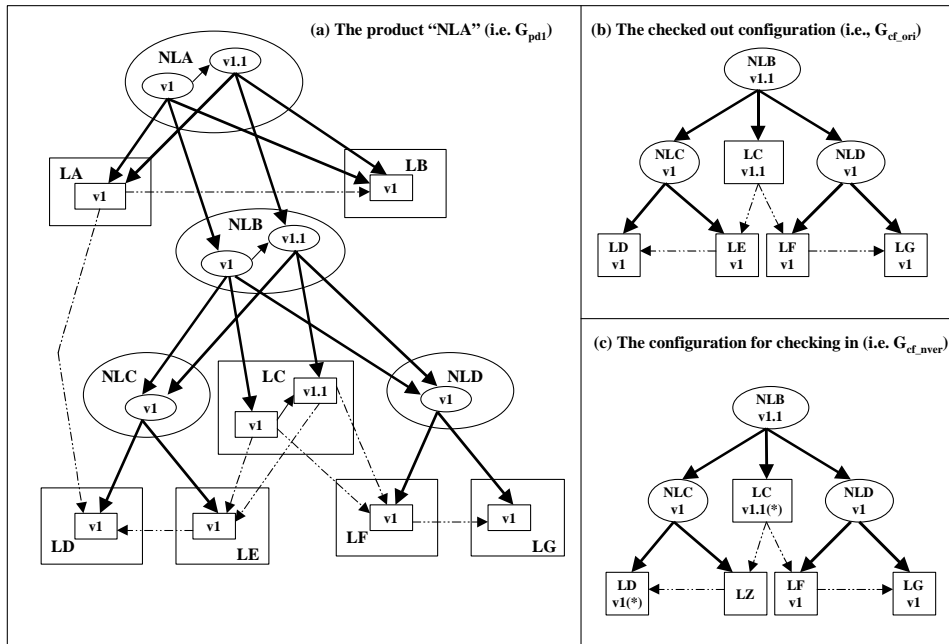


Fig. 8. Product and configuration used in the version control example.



If  $G_{cf\_ori}$  is a configuration of  $G_{pd1}$ 's sub-product, some extra intra-DEP relationships should be added between newly added revisions and nodes not in  $G_{cf\_ori}$ . For example, in Fig. 8 (a), there is an intra-DEP relationship  $\langle LA\ v1, LD\ v1 \rangle$ . It is expected that an intra-DEP relationship should be established between "LA v1" and the newly added revision "LD v1.1".

4) Create revisions for the predecessors of  $G_{cf\_ori}$ .

In our model, versions equal configurations [10]. Therefore, creating a revision for a configuration results in creating revisions for all its predecessors. For example, checking in Fig. 8 (c) to Fig. 8 (a) causes "NLA v1.1", which is the parent of  $G_{cf\_ori}$  (i.e., Fig. 8 (b)), to get a revision "NLA v1.1.1" (see Fig. 9). Then, decomposition relationships should be established between "NLA v1.1.1" and the default versions of version trees "LA", "LB", and "NLB", because they are sub-version trees of "NLA".

5) Establish inter-DEP relationships for the newly added revisions.

Inter-DEP relationships are another kind of relationships that should be established during version control. Since only developers know the referring to (i.e., inter-DEP) relationships among nodes, the developers should pass the dependency relationships to the version control process. We use Fig. 10 to explain the establishment of inter-DEP relationships. Suppose that the product "Spec" in Fig. 7 has been revised by adding new versions "subSpec2 v1.2" and "Spec v1.2" as shown in Fig. 10 (a). Also suppose that developers are required to develop a revision for "subDesign2 v1" to de-

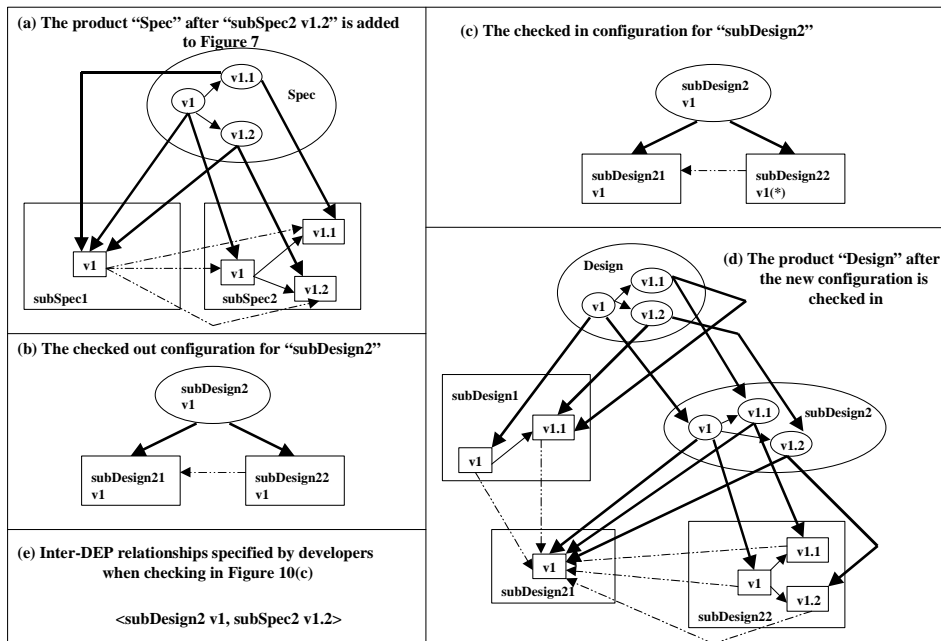


Fig. 10. Example to show the establishment of inter-DEP relationships during version control (see also Fig. 7).

pend on “subSpec2 v1.2”. The checked out design configuration is shown in Fig. 10 (b). The revision for Fig. 10 (b) is Fig. 10 (c), where “subDesign22 v1” gets a new version. After the configuration in Fig. 10 (c) is checked in, the product “Design” is revised as that in Fig. 10 (d). Suppose that the inter-DEP relationship in Fig. 10 (e) is passed to the version control algorithm when Fig. 10 (c) is checked in. Then, the inter-DEP relationship  $\langle \text{subDesign2 v1.2}, \text{subSpec2 v1.2} \rangle$  will be established (see Fig. 11), in which “subDesign2 v1.2” is the revision added for “subDesign2 v1” in Fig. 10 (d) after Fig. 10 (c) is checked in.

As described before, creating a revision for a configuration will result in creating revisions for its predecessors. Therefore, creating the revision “subDesign2 v1.2” results in the revision “Design v1.2” as shown in Fig. 10 (d). The revision “Design v1.2” must inter-depend on a node of the product “Spec”, because the product “Design” inter-depend on the product “Spec”. In our model, suppose that a new version node “A” is developed based on “B”, then the parent of “A” inter-depend on the parent of “B”. Therefore, the revision “Design v1.2” in Fig. 10 (d), which is the parent of “subDesign2 v1.2”, inter-depend on “Spec v1.2” in Fig. 10 (a), which is the parent of “subSpec2 v1.2”. According to the above description, Fig. 11 depicts the results after inter-DEP relationships have been established between the products “Spec” and “Design”.

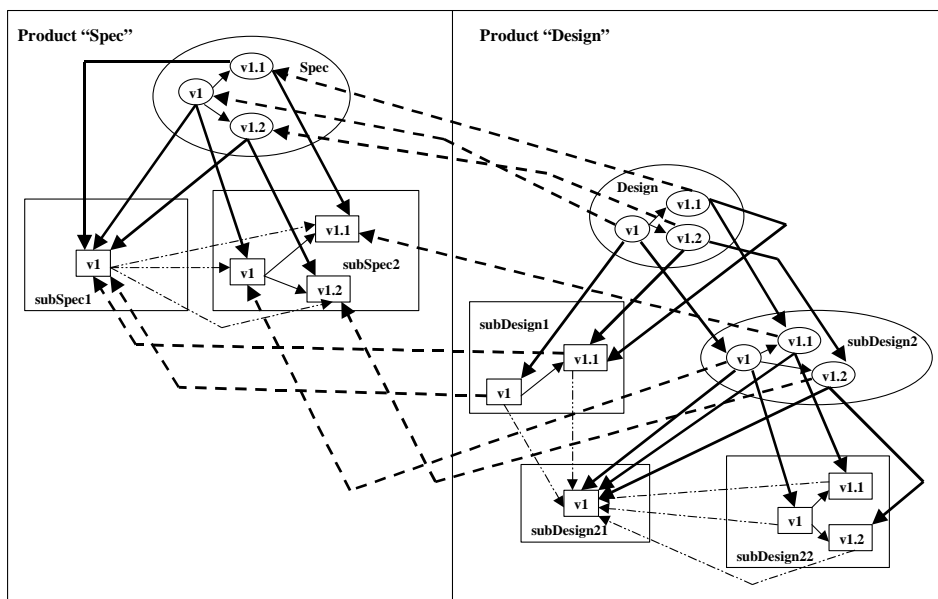


Fig. 11. The products “Spec” and “Design” after adding new inter-DEP relationships.

The above description results in the following version control algorithm. Please refer to Definitions 1 through 4 when tracing the algorithm.

**Algorithm 1** Version control

Input:

 $G_{cf\_ori} = (V_{cf\_ori}, E_{cf\_ori})$ , which is the checked out configuration; $G_{cf\_nver} = (V_{cf\_nver}, E_{cf\_nver})$ , which is the revision of  $G_{cf\_ori}$ ;/\*  $G_{cf\_nver}$  is the configuration to check in. \*/ $G_{pd1}$ , which is the product containing  $G_{cf\_ori}$ ; $E_{dep} = \{ \langle u, v \rangle \mid \langle u, v \rangle \in E_{r\_interDEP} \wedge u \in V_{cf\_ori} \wedge v \text{ is in a product other than } G_{pd1} \}$ ;

begin /\* Begin of algorithm \*/

Let  $V_{newNode} = \{ v \mid v \in V_{cf\_nver} - V_{cf\_ori} \}$ ; /\* identify new nodes \*/Let  $V_{revisionNode} = \{ v \mid v \text{ is a } revisionNode \text{ in } V_{cf\_nver} \}$ ; /\* Identify revisionNode.\*/

/\* Create a version tree for each new node. \*/

For each  $v_{new}$  in  $V_{newNode}$ , doCreate  $G_{vt\_new} = (\{ v_{new} \}, \{ \})$ , which is a version tree for  $v_{new}$ ;

/\* A version tree containing one node possesses no relationship. \*/

Let  $G_{pd1} = (V_{pd1}, E_{pd1})$ ;Modify  $G_{pd1}$  to be  $(V_{pd1} \cup \{ G_{vt\_new} \}, E_{pd1})$ ;/\* Add the newly created version tree to the product in which  $G_{cf\_nver}$  is checking back in \*/

end do;

/\* The following operations add a revision to  $G_{pd1}$  for each  $revisionNode$ .

The version tree to contain a revision node should first be identified.

Next, the revision node is added to the version tree.

Then, a version relationship is established between the revision node and its original version \*/

For each  $v_{revision}$  in  $V_{revisionNode}$ , doLet  $v_{new}$  be the newly added revision for  $v_{revision}$ ;Let  $G_{vt1} = (V_{vt1}, E_{vt1})$ , which is the version tree containing  $v_{new}$ ;Modify  $G_{vt1}$  to be  $(V_{vt1} \cup \{ v_{new} \}, E_{vt1} \cup \{ \langle u, v_{new} \rangle \mid \langle u, v_{new} \rangle \in E_{r\_version} \wedge u \text{ is the original version of } v_{new} \})$ ;

end do;

/\* When the configuration  $G_{cf\_nver}$  is checked in to the product  $G_{pd1}$ ,relationships in  $G_{cf\_nver}$  should be contained by  $G_{pd1}$ ,the following operations copy relationships in  $G_{cf\_nver}$  to  $G_{pd1}$ . \*/For each  $\langle u_{cf\_nver}, v_{cf\_nver} \rangle$  in  $E_{cf\_nver}$ , where both  $u_{cf\_nver}, v_{cf\_nver}$  are not unchanged nodes, doLet  $u_{new}$  and  $v_{new}$  be the newly added revision in  $G_{pd1}$  for  $u_{cf\_nver}$  and  $v_{cf\_nver}$ , respectively;Let  $G_{vt1} = (V_{vt1}, E_{vt1})$ , which is the version tree containing  $u_{new}$  and  $v_{new}$ ;Modify  $G_{vt1}$  to be  $(V_{vt1}, E_{vt1} \cup \{ \langle u_{new}, v_{new} \rangle \mid \text{the type of } \langle u_{new}, v_{new} \rangle \text{ is the same as that of } \langle u_{cf\_nver}, v_{cf\_nver} \rangle \})$ ;

end do;

/\* If a checked out node “n1” has an intra-DEP relationship with a node “n2” that is not checked out, the revision of “n1” should have an intra-DEP relationship with “n2”. For example, suppose the checked out node “bookMng v1” in Fig. 5 intra-depends on “borrowBook v1” in which the latter is not checked out. Then, the revision of “borrowBook v1” should intra-depends on “borrowBook v1”. The following operations establish intra-DEP relationships between newly added nodes (i.e., revisions) and nodes not in  $G_{cf\_ori}$  (i.e., nodes that are not checked out). Without loss of generality, we suppose that  $v_{pd1}$  is not checked out (i.e., not in  $G_{cf\_ori}$ ). \*/

Let  $G_{pd1} = (V_{pd1}, E_{pd1})$ ;

For each  $\langle u_{pd1}, v_{pd1} \rangle$  in  $E_{pd1}$ , where  $\langle u_{pd1}, v_{pd1} \rangle \in E_{r\_intraDEP} \wedge u_{pd1} \in V_{cf\_ori} \wedge v_{pd1} \notin V_{cf\_ori} \wedge u_{pd1}$  gets a new version in  $G_{cf\_nvers}$ , do

Let  $G_{pd1} = (V_{pd1}, E_{pd1})$ ;

Modify  $G_{pd1}$  to be  $(V_{pd1}, E_{pd1} \cup \{ \langle u_{new}, v_{pd1} \rangle \mid \langle u_{new}, v_{pd1} \rangle \in E_{r\_intraDEP} \wedge u_{new}$  is the newly added revision in  $G_{pd1}$  for  $u_{pd1} \})$ ;

end do;

/\* As described above, when a node gets a new version, every one of its predecessors will also get a new version. The following operations create new versions for predecessors of the checked out configuration  $G_{cf\_ori}$ . \*/

Let  $(v_1, v_2, \dots, v_n)$  be a sequence, in which  $v_1$  is the parent of  $G_{cf\_ori}$ 's root  $\wedge v_{i+1}$  is the parent of  $v_i$ ;

Do the followings for  $v_{parent} = v_1$  up to  $v_n$

/\* Add a revision to  $G_{pd1}$  for  $v_{parent}$ , then establish a version relationship \*/

Let  $v_{new}$  be the newly added revision for  $v_{parent}$ ;

Let  $G_{vr1} = (V_{vr1}, E_{vr1})$ , which is the version tree containing  $v_{parent}$ ;

Modify  $G_{vr1}$  to be  $(V_{vr1} \cup \{v_{new}\}, E_{vr1} \cup \{ \langle v_{parent}, v_{new} \rangle \mid \langle v_{parent}, v_{new} \rangle \in E_{r\_version} \})$ ;

/\* Establish decomposition relationships for  $v_{new}$  \*/

Let  $G_{pd1} = (V_{pd1}, E_{pd1})$ ;

Let  $S_{vr1} = \{ G_{vr1} \mid G_{vr1} \text{ is a child version tree of the version tree containing } v_{new} \}$ ;

Modify  $G_{pd1}$  to be  $(V_{pd1}, E_{pd1} \cup \{ \langle v_{new}, w \rangle \mid \langle v_{new}, w \rangle \in E_{r\_decomposition} \wedge w \text{ is the default version of a version tree in } S_{vr1} \})$ ;

end do;

/\* The following operations establish inter-DEP relationships.

The establishment is based on the input argument  $E_{dep}$ , which contains inter-DEP relationships between new version nodes and the nodes not checked out.

Since inter-DEP relationships exist between sub-products of different products, the following operations establish inter-DEP relationships between new version nodes and nodes not in the product  $G_{pd1}$  (which contains the checked out configuration).

See the “input” part of this algorithm for the definition of  $E_{dep}$ . \*/

Let  $G_{pd1} = (V_{pd1}, E_{pd1})$ ;

Modify  $G_{pd1}$  to be  $(V_{pd1}, E_{pd1} \cup \{ \langle v, w \rangle \mid \langle v, w \rangle \in E_{r\_interDEP} \wedge v \text{ is the newly added revision in } G_{pd1} \text{ for } u \wedge \langle u, w \rangle \in E_{dep} \})$ ;

/\* The following operations establish inter-DEP relationships between the predecessors of the checked out configuration (i.e.,  $G_{cf\_ori}$ ) and nodes in products other than  $G_{pd1}$ . In our model, if a new version node “A” inter-depends on “B”, then the parent of “A” inter-depends on the parent of “B”. The following operations follow the above rule to establish inter-DEP relationships \*/

```

Let  $v_{r\_cf\_ori}$  be the revision of  $G_{cf\_ori}$ 's root;
Let  $(v_1, v_2, \dots, v_n)$  be a sequence,  $v_1$  is the parent of  $v_{r\_cf\_ori} \wedge v_{i+1}$  is the parent of  $v_i$ ;
Do the followings for  $v_{parent} = v_1$  up to  $v_n$ 
  Let  $V_{son} = \{u \mid u \text{ is a son of } v_{parent}\}$ ;
  Let  $V_{dep} = \{u \mid \langle v, u \rangle \in E_{r\_interDEP} \wedge v \in V_{son}\}$ ;
  Let  $V_{parent\_dep} = \{u \mid \langle v, u \rangle \in E_{r\_version} \wedge v \in V_{dep}\}$ ;
  Let  $G_{pd1} = (V_{pd1}, E_{pd1})$ ;
  Modify  $G_{pd1}$  to be  $(V_{pd1}, E_{pd1} \cup \{\langle v_{parent}, u \rangle \mid \langle v_{parent}, u \rangle \in E_{r\_interDEP} \wedge u \in V_{parent\_dep}\})$ ;
end do;
end; /* End of algorithm */

```

To prevent keeping too many versions, versions can be merged. When versions are merged, their descendants should also be merged. For example, when “NLB v1” and “NLB v1.1” in Fig. 9 are merged, their children “LC v1” and “LC v1.1” should be merged. The merge process starts from the versions to merge down to the leaf nodes. Since descendants of the versions to merge should also be merged, the versions to merge should not have descendants belonging to different version trees. For example (see Fig. 9), “NLB v1.1” and “NLB v1.1.1” cannot be merged because they have descendants, say “LE v1” and “LZ v1”, belonging to different version trees.

After version merge, relationships among nodes should be adjusted. Generally, relationships linking to the versions before merge should be redirected to the new version obtained by the merge. If this handling results in multiple copies of the same relationships, only one copy should be kept. For example (see Fig. 9), if “LC v1” and “LC v1.1” are merged (suppose the version obtained by the merge is “LC v1.1”), then two copies of the intra-DEP relationship  $\langle LC\ v1.1, LF\ v1 \rangle$  will result, in which one of them should be deleted.

Sometimes, merging versions may result in collapsing the versions' predecessors. For example, merging the versions “NLB v1” and “NLB v1.1” in Fig. 9 causes the version tree “NLA” to collapse, because “NLA v1” and “NLA v1.1” possess the same configuration after the merge. See Fig. 12 for the result of merging “NLB v1” and “NLB v1.1” in Fig. 9. One thing we should emphasize is that the merge function does not automatically merge versions. Instead, the function identifies the versions that should be merged, then inform developers to carry out the merge.

### 3.2 Configuration Management

In the proposed model, the configuration of a version is fixed. That is, we regard versions as configurations [10]. For example, the configurations of “librarySystem v1.1” and “bookMngSubsyst v1” in Fig. 5 are respectively depicted in Figs. 6 (a) and (b).

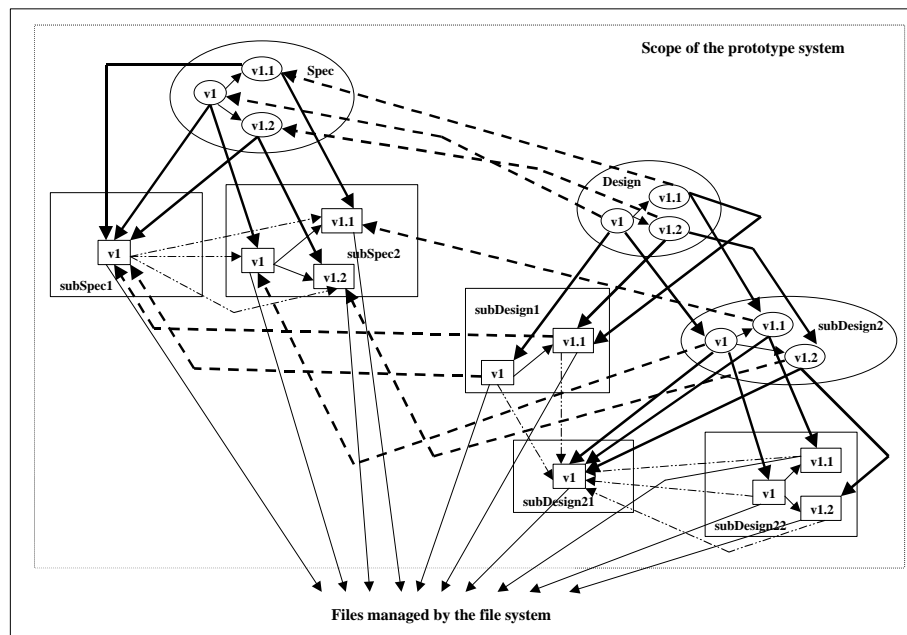


Fig. 12. The product “NLA” after merging “NLB v1” and “NLB v1.1 in Fig. 9.

To facilitate the retrieval of configurations, a configuration can be associated with the following attributes: name, version number, developers, date, paradigm (e.g., object-oriented or functional decomposition), platform, architecture (e.g., client/server architecture), programming language, operating system, software tool, the project in which the configuration was developed, and the configuration type (e.g., a specification or a design document). Moreover, each configuration can be associated with a description containing an arbitrary number of words. Among the above attributes, only the name and version number should always possess a value. With the attributes, configurations can be retrieved by: (1) name and version number, (2) default version, (3) arbitrary combination of attribute values, and (4) keywords in configuration description.

### 3.3 Product Consistency Management

Product inconsistency is normally caused by change. Consistency management refers to identifying nodes in the product graphs that are affected by a change, and requiring developers to handle them. To decrease change ripple effects, the proposed model allows a node to be divided into its *interface* and *implementation*. With this arrangement, changing a node’s implementation without changing its interface will affect fewer others. For a node that cannot be divided into those two parts, its contents should be placed in the interface part. In the following text, we first define affected nodes, then propose an algorithm to manage product consistency.

**Definition 5** If the interface of the node  $v_{changed}$  is changed, the affected nodes constitute a set  $V_{affected}$  defined below. The definition indicates that the following nodes are regarded as affected: revisions of the node, those inter- and intra-depend on the node, children of the node, and the parent of the node.

$$V_{affected} = \{v \mid \langle v, v_{changed} \rangle \in E_{r-interDEP} \cup E_{r-intraDEP} \cup E_{r-decomposition}\} \cup \{v \mid \langle v_{changed}, v \rangle \in E_{r-version} \cup E_{r-decomposition}\}$$

**Definition 6** If the interface of the node  $v_{changed}$  is not changed, changing the implementation of  $v_{changed}$  results in the following affected node set ( $V_{affected}$ ):

$$V_{affected} = \{v \mid \langle v, v_{changed} \rangle \in E_{r-interDEP} \cup E_{r-decomposition}\} \cup \{v \mid \langle v_{changed}, v \rangle \in E_{r-version} \cup E_{r-decomposition}\}$$

Changing the interface of a node will affect the nodes related to it. Therefore, Definition 5 is obvious. Definition 6 emphasizes that nodes intra-depend on a changed node will not be considered affected if the interface is not changed. For example, changing the implementation encapsulated inside a module will not affect the modules invoking it. Based on Definitions 5 and 6, Algorithm 2 depicts the process to manage product consistency.

**Algorithm 2** Product consistency management

Input:

$v_{changed}$ : the node that is changed;

begin

if the interface of  $v_{changed}$  is changed then

Identify  $V_{affected}$  as defined in Definition 5;

else

Identify  $V_{affected}$  as defined in Definition 6;

end if;

/\* Handle change ripple effects. \*/

For each  $v$  in  $V_{affected}$ , do

Require developer to determine whether  $v$  should be changed;

if  $v$  should be changed then

Require developers to change  $v$ ;

Recursively invoke Algorithm 2 to manage product consistency according to the change of  $v$ .

end if;

end do;

end;

The loop in the algorithm handles change ripple effects. Handling the effects by tracing inter-product relationships corresponds to handling inter-product change ripple

effects. On the other hand, handling the effects by tracing intra-product relationships corresponds to handling intra-product change ripple effects. Algorithm 2 thus handles both inter- and intra-product change ripple effects.

### 3.4 Reference Completeness Management

Reference incompleteness is normally caused by creating new versions. For example, in Fig. 13 (which is obtained from Fig. 7 by adding a revision “Spec v1.2” for “Spec v1”), if no new version is created for the product “Design” to inter-depend on “Spec v1.2”, then a reference incompleteness exists between the products “Spec” and “Design”. Generally, reference incompleteness indicates that some products are undeveloped. In our model, reference incompleteness is tolerable. The rationale is that a revision may be an alternative solution, which may not always implemented. Accordingly, we identify reference incompleteness and suggest developers to handle it. If the developers decide to leave the incompleteness un-handled, the incompleteness is recorded for possible handling a later time. The algorithm for reference completeness management is described below.

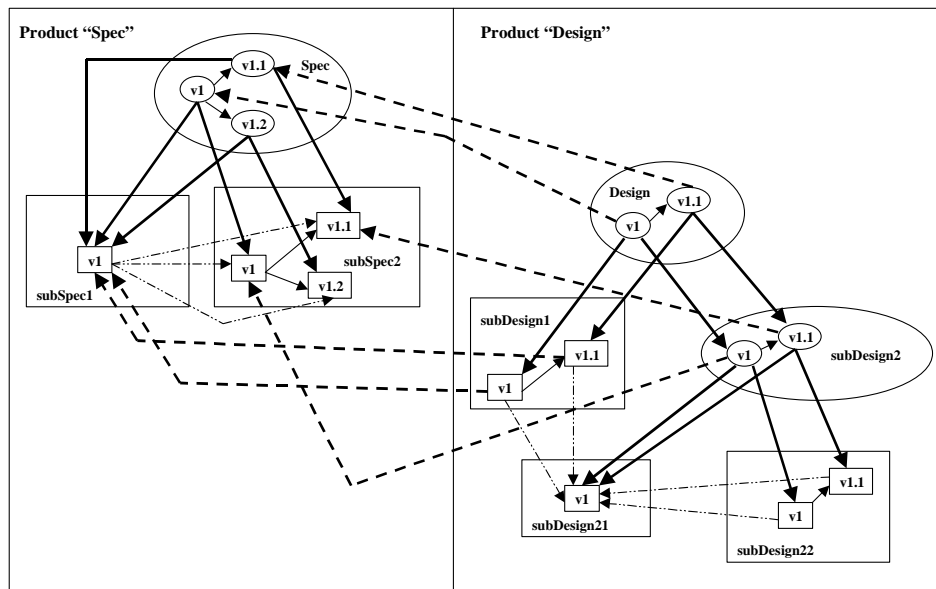


Fig. 13. Example of referential incompleteness.

#### Algorithm 3 Reference completeness management

Input:

$v_{nver}$ , which is the node to check in;

$v_{over}$ , which is the original version of  $v_{nver}$ ;

begin

Let  $V_{dep} = \{v \mid \langle v, v_{over} \rangle \in E_{r-interDEP}\}$ ;

```

For each  $v$  in  $V_{dep}$ , do
  Suggest developers to develop  $v_{new}$ , which is a revision of  $v$ , to inter-depend on  $v_{nver}$ ;
  if the suggestion is refused, then
    Record the reference incompleteness;
  else
    Recursively invoke Algorithm 3 to manage reference incompleteness according
    to the new version  $v_{new}$ ;
  end if;
end do;
end;

```

### 3.5 Product Reuse Support

Product reuse refers to reusing existing products (or sub-products) to develop new ones. Generally, techniques are needed to classify existing products, to retrieve similar products, and to compose the reused products. If products are represented in our model, those being reused are configurations.

After the reuse of a configuration, various *correlated configurations* as mentioned in section 1 may be considered reusable. We use Fig. 14 and the following description to re-visit the concept of reusing correlated products (i.e., correlated configurations)

- 1) Suppose the specification configuration “JournalMngSpec v1” in Fig. 14 is reused. Then, by tracing inter-DEP relationships, the design configurations “JournalMngDesign v1” and “JournalMngDesign v1.1” are candidate correlated configurations for reuse, because they are the design documents of “JournalMngSpec v1”.
- 2) Suppose the design configuration “ReturnBookMng v1” is reused. Then, by tracing decomposition relationships, the design configurations “BorrowBookMng v1”, “BookMngDesign v1”, and “JournalMngDesign v1” are candidate correlated configurations for reuse, because sub-configurations belonging to the same configuration may be reused together.
- 3) Suppose the specification configuration “BookMngSpec v1” is selected for reuse. Then the specification configurations “BookMngSpec v1.1” and “BookMngSpec v1.2” can be considered alternative selections for reuse (here alternative selections are also correlated configurations).

The above description defines three types of correlated products. According to the descriptions, the correlated configurations ( $S_{cf\_corr}$ ) and the alternative selections ( $S_{cf\_alt}$ ) are defined below, in which  $G_{cf\_reused}$  is the reused configuration,  $S_{cf\_corr}$  covers the first two types of correlated products, and  $S_{cf\_alt}$  covers the third type of correlated products.

#### Definition 7

$$\begin{aligned}
S_{cf\_corr} &= \{G_{cf1} \mid G_{cf1} \text{ is a configuration whose root inter-dependes on } G_{cf\_reused}'\text{'s root}\} \\
&\cup \{G_{cf1} \mid G_{cf1} \text{ is a configuration whose root and } G_{cf\_reused}'\text{'s root possess the same} \\
&\text{predecessor(s)}\} \\
S_{cf\_alt} &= \{G_{cf1} \mid G_{cf1} \text{ is a configuration whose root and } G_{cf\_reused}'\text{'s root are in the same} \\
&\text{version tree}\}
\end{aligned}$$

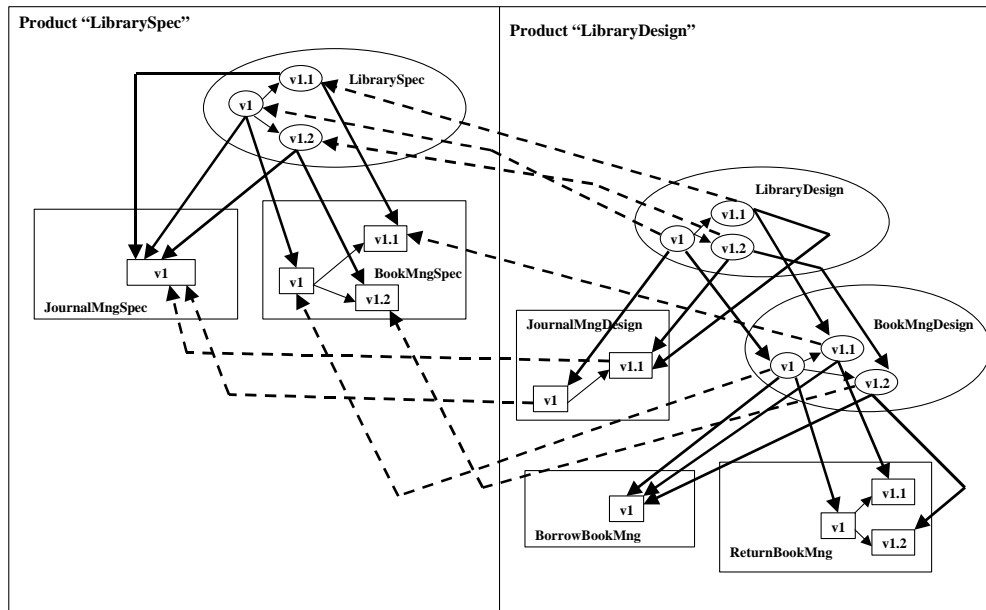


Fig. 14. The products "LibraryDesign" and their relationships.

Definition 7 reveals that our model facilitates identifying correlated configurations for reuse. Generally, identifying correlated configurations should start after a reuse. That is, the identification should be embedded in a technique that retrieves reusable software components. We have developed a technique to reuse correlated configurations based on the proposed product model [25].

#### 4. IMPLEMENTATION AND DISCUSSIONS

We have implemented a prototype system for the product model and product management functions. The prototype is implemented in C language using Borland C++ builder. It is about 3K lines in length (much of the code handles user interface). As shown in Fig. 15, the system manages nodes of product hierarchies and relationships among nodes. A leaf node, which corresponds to a real document, records the name of file that stores the document. The leaf also indicates the location of the document in the file, because documents represented by multiple leaves may be placed in the same file (e.g., documents in the leaves of Fig. 2 may be placed in the same file). We implemented the system in this manner because different products may be associated with different tools (e.g., ROSE, MS Word, and C++ Builder) that cannot be controlled by the prototype system. With this implementation, accessing a leaf results in opening the corresponding file by the associated tool. This simplifies the implementation.

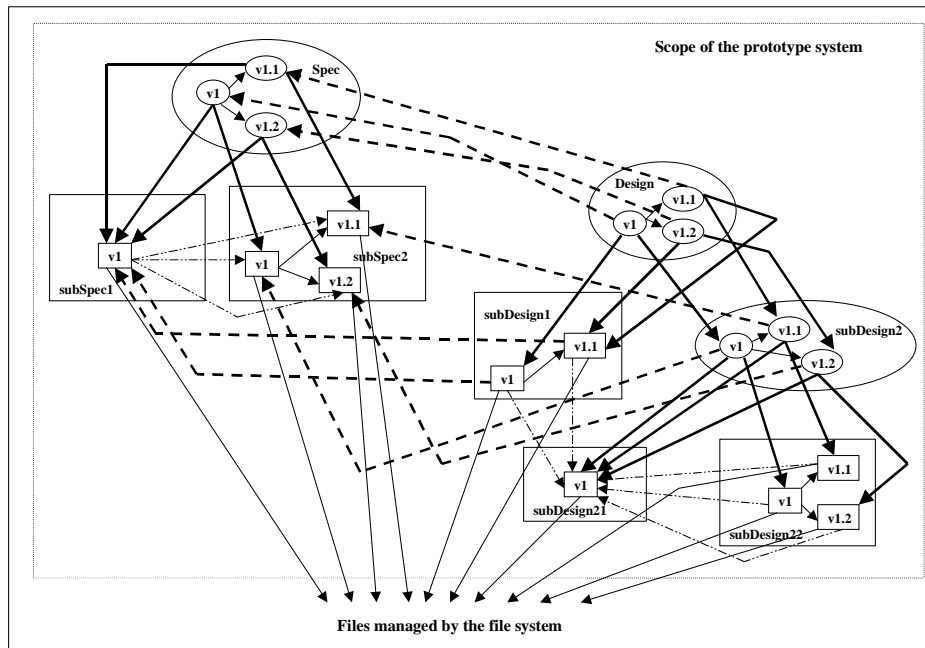


Fig. 15. Scope of the prototype system.

Currently we place 12 products in the system. They are the specification, design document, and program code of a simplified library system, those of a simplified super-market management system, those of a simplified student grade management system, and those of a simplified book store management system. According to our experiences in using the system, we give some discussions below.

#### 1) Performance

We discuss the time complexity of the algorithms here. As to the disk space used, it is evaluated in the following two items.

The worst case of time needed by the version control algorithm (i.e., Algorithm 1) is dominated by the establishment of intra-DEP relationships between the newly added nodes and nodes not in  $G_{cf\_ori}$ . The time complexity of the establishment is  $O(i*j)$ , where  $i$  is the number of nodes in  $G_{pd1}$  and  $j$  is the number of nodes in  $G_{cf\_ori}$ . The worst case will occur only when every node not in  $G_{cf\_ori}$  has a relationship with every node in  $G_{cf\_ori}$ . But this is seldom the case. According to our experiences, the average time complexity is  $O(j^2)$ . Since developers tend not to check out a configuration with too many nodes, the value of  $j$  tends to be small. Therefore, the version control algorithm is expected to be efficient.

The time complexity of Algorithm 2 cannot be analyzed, because the number of nodes affected by a change cannot be predicted. In our prototype system, changing a node will affect seven nodes in average. Nevertheless, we cannot give any conclusion to

the time complexity of Algorithm 2, because the prototype system currently manages only 12 products and more important, the product sizes are small.

The time complexity of Algorithm 3 is  $O(n)$ , where  $n$  is the number of products being managed. The rationale is that reference incompleteness is induced by inter-DEP relationships and at most  $n$  nodes can appear in a chain linked by inter-DEP relationships.

## 2) Version proliferation

As described in section 3.1, when a new version of a configuration is checked in, every predecessor of the configuration (identified according to decomposition relationships) will get a new version. If the hierarchy of a product is high, versions may proliferate seriously. We use an example to explain this. Suppose that the height of a product is  $n$  and each non-leaf has  $k$  children in average. Then, in case that every node of the product gets a new version, the product will get  $m$  new versions, in which

$$m = (k^n - 1)/(k - 1) + k*(k^{n-1} - 1)/(k - 1) + \dots + k^{n-1}$$

Note that the value “ $(k^n - 1)/(k - 1)$ ” corresponds to the new versions obtained by the root of the product, the value “ $k*(k^{n-1} - 1)/(k - 1)$ ” corresponds to the new versions obtained by the children of the root, and so on. In a model that will not cause version proliferation, only  $(k^n - 1)/(k - 1)$  new versions will be obtained in the above case. Comparing this value with  $m$ , version proliferation does result in many versions to manage.

There are pros and cons associated with version proliferation. For the former, configuration management is easy. For the latter, much disk space may be needed to store versions. Let’s evaluate the space wasted according to version proliferation. As described in the beginning of this section, our implementation puts real documents in files. As to nodes managed by the system, they store their information (e.g., name, version number, and description of the node) and the relationships associated with the nodes. Currently, we reserve 1K bytes for each node. For a product hierarchy with a height of  $n$ , adding a new version to a leaf node will cause the hierarchy to obtain  $n$  new versions. Therefore, in the worst case, our model manages extra  $n - 1$  new versions when a node gets a new version. Suppose that  $n$  equals to 10 (note that a product hierarchy with a height of 10 is *very high*), our model wastes 9K bytes of disk space at worst to manage extra new versions when a new version is added. To our opinion, this space wasting is not that serious as imaging. Nevertheless, we still think that version proliferation should be solved. We leave the problem as a future work.

## 3) Delta

The proposed model does not incorporate the delta technique [6, 7] to store versions. That is, every version is stored in its entirety, with which disk space may be wasted. Let’s use Fig. 2 to explain this. The figure depicts a program consisting of five modules and the corresponding product hierarchy has five leaves (note that the number of modules needs not always equal to the number of leaves because several modules can be represented by a leaf). If the program modules are placed in the same file, any of the modules gets a new version will result in a new file. This does result in disk space wasting. Let’s image that a program occupies 100K bytes of disk space and the program possesses

50 modules. Then, adding a new version for every module results in 51 versions of the file, which occupy 5.1M bytes of disk space. If the products being managed are specifications or design documents, the document size should be far larger than 100K bytes. In this case, far more disk space will be wasted in storing versions of leaves. Therefore, incorporating the delta technique seems anxious for the proposed model.

Saving space is not the only advantage of the delta technique. We have experienced the potential advantage of the delta technique in version merge. As described in section 3.1, the version merge function identifies versions that should be merged, then requires developers to manually merge the versions. Some students complained that identifying the difference among versions takes much time. In this case, if the delta technique is used, the difference can be automatically identified. According to the advantages above, incorporating the delta technique is another future work to upgrade our model. Nevertheless, applying the delta technique to the model is not easy because the following issues should be solved (remember that accessing a leaf of the model results in opening the corresponding file by the associated tool):

- a. Understand the data format of every possible tool. Without understanding the data format, delta (i.e., difference) between two versions cannot be correctly identified.
- b. Apply delta information to a file before the file is opened by the corresponding tool.

#### 4) Granularity of product decomposition

Fine-grained decomposition of a product results in deep (or high) product hierarchy and small-sized leaf nodes. In a deep product hierarchy, version proliferation is serious, as has been discussed. Nevertheless, small-sized leaves improve traceability. We use an example to explain this. Suppose that a sub-product of a product is changed. If the decomposition of the product is coarse-grained, the sizes of the affected sub-products identified by Algorithm 2 would be large. According to our experiences, identifying the parts that are indeed affected from a large-sized product spends much time. On the other hand, if the decomposition is fine-grained, the sizes of the affected sub-products will be small. Identifying the parts that are indeed affected from a small-sized product spends less time.

Fine-grained decomposition also facilitates reducing disk space for versions. As discussed previously, if every module in a 100K program (which possesses 50 modules) gets a new version, 5.1M disk space is needed for versions. To reduce disk space needed, fine-grained decomposition can be used. For example, we can decompose the 100K program until every leaf in its product hierarchy consists of one module only, and then store each module of the program in a file. With this arrangement, about 200K disk space is needed for versions in case that every module gets a new version. Comparing 200K with 5.1M, it is a dramatic saving. Currently, we are using fine-grained decomposition to implement the delta technique.

## 5. RELATED WORK

Since our model emphasizes product relationships that can facilitate product management, our survey primarily focuses on product relationships modeling.

SCCS [6] manages document versions using a tree structure. When a changed

document is checked in, a new version is created for it. In storing versions, the delta technique is used. SCCS does not decompose documents. Therefore, neither decomposition nor intra-DEP relationship is managed. Moreover, no inter-DEP relationship is managed.

RCS [7] manages document versions using a tree structure. The delta technique is used to store versions. It does not decompose documents. Therefore, neither decomposition nor intra-DEP relationship is managed. As to inter-DEP relationships, RCS uses configuration management to bind related documents. To select a configuration, attributes such as version number, date, and name can be used. RCS allows concurrently changing a document, in which each change result in a new version. New versions should be merged later.

POEM [8] manages functions and classes inside a program. Therefore, POEM regards a program as a collection of components. Intra-DEP relationships including *uses\_interface* and *t\_uses\_interface* are used to structure program components. Only one layer of decomposition is used. The RCS-like tree is used to structure versions. During version control, a component will obtain new versions if the components it uses get new versions. The “versions equal configurations” approach is used for configuration management and therefore versions may proliferate. Since POEM manages components of a single program, no inter-DEP relationship is managed.

NUCM [9] manages versions of artifacts, which are either atoms (documents or code) or collections of artifacts. It maintains containment relationships between a collection and the artifacts it contains. Although a collection can contain various kinds of documents such as code and documents for the code, there is no inter-DEP or intra-DEP relationship established for artifacts. New versions can be created for an artifact. To prevent version proliferation, new version for an artifact will not result in new versions for the artifacts containing it. No version relationship is used to structure versions of the same artifacts. Explicit version numbers should be used when retrieving a configuration (collection).

The Ragnarok architectural SCM model [10] manages software and their versions. A software system can be decomposed into components, and intra-DEP relationships can be established among the components. Changing components results in new versions, which are structured by the traditional version graph such as the RCS tree structure. The model uses the “versions equal configurations” approach for configuration management. Therefore, versions may proliferate. Versions can be merged. However, the versions being merged still exist after the merge. The model manages versions and relationships inside a single software product. No inter-DEP relationship is managed.

The model according to Lindsay [11] proposes a fine-grained configuration management approach. The model manages documents developed during software development and manages inter-DEP relationships among sub-products of different products. Therefore, the traceability among products is well-maintained. However, the model does not maintain intra-DEP relationships. Therefore, the traceability inside a product is not maintained. Version control is on the sub-product (paragraph) level. That is, each paragraph can be versioned. Version trees are used to structure versions. Conformance matrixes (like cross reference matrixes) are used to retrieve configurations.

ClearCase [12] manages versions of all kind of files (not just text files) as well as directories. Both directories and files can have multiple versions. Versions are structured

by a tree structure. Delta is used in storing versions. Developers can check out an element (file or directory) into a view for change, then checks it back in to form a new version. The relationships managed by ClearCase are version relationships and the relationships between directories and sub-directories (or files), in which the latter are similar to the containment relationships in NUCM. No decomposition is applied to a file. Therefore, no decomposition and intra-DEP relationship is established. Although ClearCase manages versions of multiple products, it seems that no inter-DEP relationship is managed.

The AVL dags [13] model represents a product using a generalization of AVL tree. Nodes in a tree are lines (parts) of a product. A product can be reconstructed by in-order traversing its corresponding AVL tree. When a revision is created for a node of an AVL tree, revisions will be created for the node's predecessors. Therefore, versions may proliferate. Although the model divides products, no intra-DEP relationship is managed. Moreover, the model does not manage inter-DEP relationship.

The HiP model [14] also uses a tree to structure a product. Versions of a tree are kept in a version control tree. Information of the change history of a node is saved in the node's parent. Therefore, previous version can be reconstructed from the current version. Like AVL dags model, HiP model does not manage inter-DEP and intra-DEP relationships.

DSEE [15] controls versions of source code and the corresponding derived objects. A system model is used to describe the components of a system. Components can be versioned. Delta is used in storing versions. Configuration threads (rules-based descriptions) are used to retrieve configurations. Inter-DEP relationships are established between systems and derived objects. Although systems can be decomposed into components, no intra-DEP relationship is established among the components of a system.

Adele 2 [16, 22] is composed of a multi-versioned Adele DB and multiple single-versioned work environments. Versions are managed using version branch [16]. Adele-2 manages products developed during software development. Products are structured using a directory structure. Therefore, it manages relationships across products. However, product decomposition is not modeled. Therefore, neither intra-DEP nor decomposition relationships is managed.

EPOS [23] stores meta-classes, classes, instances of classes, and instances of software process models in EPOSDB. Contents in EPOSDB are all subject to version. Versions can be merged. Since products developed during software development are managed, relationships across products are managed. However, like Adele 2, product decomposition is not modeled. Therefore, neither intra-DEP nor decomposition relationships is managed.

CVS [24] stores versions of source program files in a centralized repository. Delta is used in storing versions. When a program file is checked out, CVS creates a copy of the file for necessary change. CVS allows multiple developers to check out the same program file concurrently. If concurrent check-out occurs, CVS automatically merges the check-in files after every checked out files have been checked back in. Since CVS manages source programs only, no inter-DEP relationship is established. Since no decomposition is allowed for a file, CVS does not manage intra-DEP and decomposition relationships.

In addition to the models described above, we also surveyed various version control models [17-21], which manage versions inside a single product. All those models do not

manage inter-DEP relationships.

When comparing with the models we surveyed, we have identified the following strength and weakness of our model:

- 1) Our model manages product relationships including version, decomposition, inter-DEP, and intra-DEP relationships. They are useful in product management such as consistency management, reference completeness management, and product reuse support.

According to our survey, we cannot locate a model that manages all those relationships. For example, version relationships are not managed in NUCM. Inter-DEP relationships are not managed by SCCS, RCS, POEM, ClearCase, HiP, the Ragnarok architectural SCM model, CVS, and so on. Intra-DEP relationships are not managed by most of the surveyed models except POEM and the Ragnarok architectural SCM model. And, decomposition relationships are not managed by SCCS, RCS, ClearCase, CVS, and so on.

- 2) Our model allows dividing a product into its interface and implementation, which can reduce ripple effects when a product is changed. In the surveyed models, we have located only POEM offers this feature.
- 3) Just like the POEM, AVL dags, and the Ragnarok architectural SCM model, versions may proliferate in our model. Although we provide version merge function, version proliferation problem should still be solved. Solving the problem is a future work to upgrade our model.
- 4) Our model does not use the delta technique to store versions. This may result in wasting disk space and ill support of version merge. Incorporating the delta technique is another future work for our model.

## 6. CONCLUSIONS

This article proposes a product model that emphasizes product relationships, and describes product management using the model. Relationships modeled include versions, decomposition, inter-DEP, and intra-DEP relationships. The relationships maintain product traceability, with which product management functions such as consistency management, reference completeness management, and product reuse (i.e., software reuse) support can be facilitated. The proposed model and functions offers the following features:

- 1) The version, decomposition, and intra-DEP relationships facilitate handling intra-product change ripple effects. Moreover, inter-DEP relationships facilitate handling inter-product change ripple effects. Accordingly, product consistency management is well-supported.
- 2) The version and inter-DEP relationships facilitate managing reference completeness. That is, when a new version of a configuration is created, inter-DEP relationships can be traced to identify reference incompleteness.
- 3) Inter-DEP and decomposition relationships can be traced to identify correlated configurations for possible reuse when a configuration is reused. In addition, version re-

- relationships can be traced to identify alternative selections for reuse.
- 4) A version merge function is provided for the model to prevent keeping too many versions.
  - 5) The model allows dividing a product into its interface and implementation. This reduces change ripple effects when a product is changed.

### ACKNOWLEDGMENT

This research is sponsored by the National Science Council in Taiwan under Grant Number NSC89-2213-E-259-012. Special thanks are given to Professor S. L. Peng in National Dong Hwa University for his valuable comments.

### REFERENCES

1. J. Y. Chen and S. C. Chou, "Consistency management in a process environment," *Journal of Systems and Software*, Vol. 47, 1999, pp. 105-110.
2. J. C. Grundy, J. G. Hosking, and W. B. Mugridge, "Supporting flexible consistency management via discrete change description propagation," *Software-Practice and Experience*, Vol. 26, 1996, pp. 1053-1083.
3. P. Freeman, *Tutorial: Software Reusability*, IEEE Computer Society Press, 1987.
4. S. C. Chou, J. Y. Chen, and C. G. Chung, "A behavior-based classification and retrieval technique for object-oriented specification reuse," *Software-Practice and Experience*, Vol. 26, 1996, pp. 815-832.
5. D. J. Chen and P. J. Lee, "On the study of software reuse using reusable C++ components," *Journal of Systems and Software*, Vol. 20, 1993, pp. 19-36.
6. M. J. Rochkind, "The source code control system," *IEEE Transactions on Software Engineering*, Vol. 1, 1975, pp. 364-370.
7. W. F. Tichy, "RCS – a system for version control," *Software-Practice and Experience*, Vol. 15, 1985, pp. 637-654.
8. Y. J. Lin and S. P. Reiss, "Configuration management with logical structures," in *Proceedings of 18th International Conference on Software Engineering (ICSE)*, 1996, pp. 298-307.
9. A. van der Hoek, D. Heimbigner, and A. L. Wolf, "A generic, peer-to-peer repository for distributed configuration management," in *Proceedings of 18th International Conference on Software Engineering (ICSE)*, 1996, pp. 308-317.
10. H. B. Christensen, "The Ragnarok architectural software configuration management tool," in *Proceedings of Thirty-second Annual Hawaii International Conference on System Science (HICSS)*, Vol. 8, 1999.
11. P. Lindsay, Y. Liu, and O. Traynor, "A generic model for fine grained configuration management including version control and traceability," in *Proceedings of Australian Software Engineering Conference*, 1997, pp. 27-36.
12. *ClearCase Concepts Manual*, Rational, 1998.
13. C. Fraser and E. Myers, "An editor for revision control," *ACM Transactions on Programming Languages and System*, Vol. 9, 1986, pp. 277-295.
14. E. J. Choi and Y. R. Kwon, "An efficient method for version control of a tree data

- structure,” *Software-Practice and Experience*, Vol. 27, 1997, pp. 797-811.
15. D. B. Leblang and R. P. Chase, Jr., “Parallel software configuration management in a network environment,” *IEEE Software*, Vol. 4, 1987, pp. 28-35.
  16. N. Belkhatir, J. Estublier, and W. L. Melo, “Software process model and work space control in the Adele system,” in *Proceedings of the 2nd International Conference on Software Processes*, 1993, pp. 2-11.
  17. The GOODSTEP Team, “The GOODSTEP project: general object-oriented database for software engineering processes,” in *Proceedings of Asia Pacific Software Engineering Conference*, 1994, pp. 410-419.
  18. C. L. Chee and S. S. Erdogan, “An installable version control file system for Unix,” *Software-Practice and Experience*, Vol. 27, 1997, pp. 725-746.
  19. A. W. Lee and H. J. Kim, “Object versioning in an ODMG-compliant object database system,” *Software-Practice and Experience*, Vol. 29, 1999, pp. 479-500.
  20. D. Lieuwen and N. Gehani, “Versions in Ode: Implementation and experiences,” *Software-Practice and Experience*, Vol. 29, 1999, pp. 379-416.
  21. J. Rho and C. Wu, “An efficient version model of software diagrams,” in *Proceedings of Asia Pacific Software Engineering Conference*, 1998, pp. 236-243.
  22. N. Belkhatir, J. Estublier, and W. L. Melo, “Adele 2: a support to large software development process,” in *Proceedings of the 1st International Conference on Software Processes*, 1991, pp. 159-170.
  23. M. L. Jaccheri and R. Conradi, “Techniques for process model evolution in EPOS,” *IEEE Transactions on Software Engineering*, Vol. 19, 1993, pp.1145-1156.
  24. P. Cederqvist, “Version management with CVS,” <http://ftp.cvshome.org/cvs-1.11.3/cederqvist-1.11.3.pdf>.
  25. S. C. Chou, “Retrieving correlated software products for reuse,” *IEICE Transactions of Information and Systems*, Vol. E87-D, 2004, pp. 175-182.



**Shih-Chien Chou (周世杰)** received a Ph.D. degree from the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan. He is currently an associate professor in the Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan. His research interests include software engineering, process environment, software reuse, and information flow control.

**Chun-Wei Huang (黃俊瑋)** received a master degree from the Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan. He currently serves as a soldier in Chinese Army.