Managing Product Evolution in Agile Manufacturing Environments

Min. Jin* · T. C. Ting**

⟨Abstract⟩

This paper presents an integrated object-oriented database approach for managing the evolution of products in agile manufacturing environments. Schema evolution modification facilities are provided to support full potential versioning of type definitions. All of the possible versions for a composite product are not explicitly represented to avoid version proliferation. However, valid configurations of any composite products can be provided to comply with customer demands. The attributes of composite products are classified in order to provide well-defined representation scheme for composite products and to be exploited in version control. The attributes are partitioned into composite-related and non composite-related categories. Composite-related attributes consist of subproducts and description ones. Subproducts attributes represent physical constituents of a composite product. Description attributes represent external features, assembling, and correspondence property. Interface attributes are introduced for managing configurability and version propagation. Version derivations due to the changes to the interface attributes are propagated toward the product composition hierarchy. The validity of configurations of composite products is checked by using configurability maps. Instance objects which represent the actual product instances are stored and manipulated in the database in order to support traceability during product life cycle.

* Dept. of Computer Sci. & Eng. Kyungnam University

** Dept. of Computer Sci. & Eng. University of Connecticut

1. Introduction

As technology continues to develop, both the engineering products and the processes which produce these products have become increasingly complicated. There are more competition among manufacturing enterprises. They are forced to change manufacturing environments. Hence, manufacturing environments move to agile manufacturing. In agile manufacturing, enterprises respond rapidly to changes in customer demands and market environments in order to provide high quality and highly customized products by exploiting agility. Product evolution is the key factor in agile manufacturing environments. Products evolve due to several reasons; fixing faults or errors, getting improvements, providing customer requirements, and exploiting new technology. Maintenance facility is also required in order to provide service and to get feedback from the customers. The data dealt with in such agile manufacturing environments are complex. Features of such data can not be characterized as a set of simple attributes. Products have complex structures. These complex structures should be represented in well-organized manner in order to accelerate the processing of design and manufacturing. Most of the data also have composite structure. A product is comprised of several constituents, each of them might be comprised of other constituents recursively. Composite hierarchies are needed for representing such products. Relational database systems have been much developed and widely spread in computing environments. However, conventional relational database systems have been proved inadequate for emerging database application areas such as computer-aided design, manufacturing, and software engineering, and office automation. Conventional relational data models have serious drawbacks in representing and manipulating complex data structures since each attribute should be atomic. Object-oriented data models appear to be attractive in design and manufacturing environments in which large and complex products are designed and produced. Core object-oriented technologies are characterized as four important concepts; objects, encapsulation, inheritance, and polymorphism. IS-PART-OF hierarchy and the notion of objects are convenient features for representing complex and composite products.

In this paper, an object-oriented database approach for managing versions and configurations of evolving products in agile manufacturing environments is proposed. Our approach is based on the schema evolution[2, 3, 6, 8, 11, 13, 15, 19, 20] in multiple version environments. A version is a variation of a type definition, which reflects changes to the product. Products which were created under different versions of the product types are also stored in the database and manipulated to provide traceability during product life cycle.

The remainder of this paper is organized as follows. In section 2, we review the research which has been done in related areas. The research has been focused on versioning of design objects in engineering database applications. In section 3, we discuss the representation mechanisms for evolving composite products including attribute classification and three hierarchies: IS-A, IS-PART-OF, and IS-VERSION-OF hierarchies. In section 4, we discuss version control mechanism. Schema modification taxonomy is mentioned. An anchor type is created whenever the first version of a type is derived. Changes to interface attributes are propagated along the product composition hierarchy. In section 5, configuration management issues are discussed. Configurability maps are introduced to check the validity of configurations. Registered configurations are kept to be used in the derivation and construction of configurations. In Section 6, we raise further research issues and offer conclusions.

2. Related Work

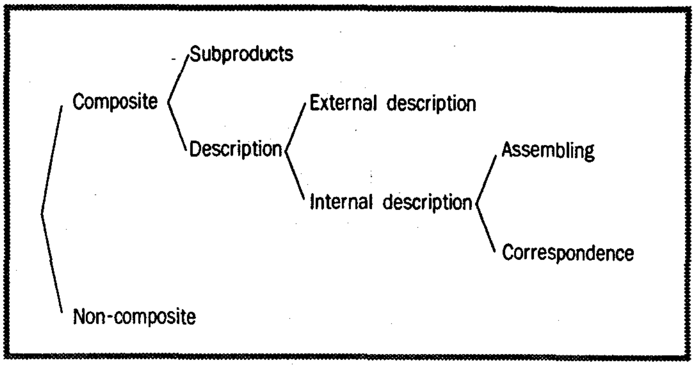
Various schemes of version control in engineering databases have been proposed and implemented[1, 4, 5, 7, 10, 13, 16]. Ditrich and Lorie[7] present a model of versions based on the concepts of design objects, generic references, hierarchical environments, and logical version clusters. This model is based on relational database systems. References can be direct or generic. Generic references are used to provide dynamic configurations and resolved by means of hierarchical environments. Versions can be clustered according to the criteria given by the user. Klahold et al. [13] propose a model based on the concept of a version environment which consists of a set of object and a set of version structures which are based on two mechanism: graphs and partitions. They also provide views of the version graph, which are subsets of the graph satisfying certain criterion. Batory and Kim[4] propose a version model based on E-R model extended with object-oriented inheritance. The model has four major concepts: molecular objects, type-version generalization, instantiation, and parameterized versions. There is explicit distinction between interfaces and implementations in molecular objects. Molecular objects are used for describing complex objects. Interfaces are used as specification of an object type, and versions are used as implementations of the same interface of the object type. Instances are distinct from versions in the concept of instantiation. An instance means an actual usage and a copy of version, whereas a version means a definition. Parameterized versions support dynamic configurations so that a version of component can be assembled in a molecular object. They also address change notification based on timestamp information. It helps limit the range of change notification messages which the system must send. Beech and Mahbold[5] propose a version model based on an object-oriented database system, IRIS. A version is an object with its own unique object identifier. Version instances of an object are organized into a version set which is associated with a generic instance of the object. The generic instance has the information on the versions and the relationships among the versions. Versions can be created in two ways; implicitly due to propagation and explicitly due to mutation. The context mechanism is introduced to specify when and how generic references are to be resolved. A context is an object which contains trigger, predicate, and action. Rumbaugh[16] proposes a mechanism for controlling operation propagation across relationships and related objects. A relation is used as a semantic construct that represents association between objects of classes. Propagation attributes on the relationships between objects specify the manner in which operations are propagated. Propagation attribute values are associated with the combination of the followings; operation name, object class, and the relationship to another object. Sciore[17, 18] propose high level application independent versioning scheme. Version sets can be viewed in multidimensional spaces. It has strong expressive power and provides flexible configurations. It adopts the concept of contexts for default version and allows combining contexts and explicit references to versions. Ahmed and Navathe[1] propose version management scheme for composite objects. Attributes of composite objects are classified into external features and internal assembly. A notion of version equivalence, which is based on the generic types, invariant external features, and value acquisition, is introduced to prevent version proliferation.

The research has been focused on versioning design objects in design environments. They have not paid attention to manufacturing environments. Moreover, the traceability issues were not taken into consideration in the above version models. In this paper, an object-oriented database approach for managing product evolution in agile manufacturing environments is proposed. Even though the traceability issue is not dealt with in this work, it is taken into considera-tion in the development of the approach to be incorporated in the further work.

3. Representation of Composite Products

3.1 Attribute Classification

〈Figure 1〉 Attribute classification of composite products



The attributes of composite products are classified into composite-related and non composite-related categories. Non composite-attributes represent associative description such as production company name of a product type. Composite-related attributes are divided into subproducts and description attributes. Subproducts attributes represent physical constituents of a composite product such as assemblies, subassemblies, or parts. Description attributes are divided into external and internal description attributes. External attributes represent external features such as interfacing association, shape, size, material, and etc.. Internal description attributes are divided into assembling and correspondence attributes. Assembling attributes represent interconnection relationships among physical constituents. Correspondence attributes represent relationships between interface of a composite product and its subproducts. Subproducts and external description attributes can be designated as interfaces of the composite product. Interface attributes are used in version propagation and change notifications. Version derivation due to the changes to interface attributes are notified to be propagated along the product composition hierarchy.

3.2 IS-A Hierarchy

Object-oriented systems allow a type(class) to have any number of subtypes(subclasses). A supertype implies generalization of its subtypes, whereas a subtype implies specialization of its supertype. All the properties of a supertype are inherited to its subtypes unless they are refined in the subtypes, so that it is called inheritance hierarchy. Inheritance hierarchy has a few advantages over database systems. First, it provides strong modeling power, it gives us an accurate and simplified description of the real world. It helps us factor out common properties among entity types in its supertype. Second, it helps code reusability and extensibility. Objects that have common operations can share the same methods which are defined in higher level class. Third, it helps lessen the burden of programmers by reducing the size of codes. We distinguish "abstract types" from "concrete types". It helps reduce the complexity of version handling and configuration management. An abstract type means a type that can not have direct instance objects. An IS-A node in the type hierarchy has the property of exclusive or"(we call this as XOR node) in the selection of a configuration of a product. An XOR node does not need to have configurability maps, which will be discussed later. When a version of XOR node(supertype) is determined in the selection of a configuration of a product, one of its children nodes(subtypes) should be selected.

3.3 IS-PART-OF Hierarchy

IS-PART-OF hierarchy is essential to representing composite objects[12]. IS-PART-OF hierarchy has AND property, whereas IS-A hierarchy has XOR property. In actual design and manufacturing environments, we have mandatory and optional constituents for a composite product. Mandatory constituents are necessary and optional constituents are optional in the composition of the products. In version level representation, there is distinction between them, in configuration level in which all the characteristics of a certain product are determined, there is no such distinction between them. The configuration specification process for composite products includes determining versions and selecting optional components.

3.4 IS-VERSION-OF Hierarchy

Versions are used for representing historical record, revisions, and alternatives of evolving products in agile manufacturing environments. We need a hierarchy of versions to keep the version history and the relationship among versions. This hierarchy is used in derivation of new versions. This is also referenced in the construction of configurations of composite products. Even though IS-VERSION-OF relationship can be constructed in an inheritance hierarchy, we do not allow inheritance since there is no relationship like generalization and specialization among versions of a product type as in IS-A hierarchy and schema are different from conventional versioned files in which delta technique that is a kind of selective inheritance is used. Since the definition of each version is frequently used in the derivation of versions and construction of configurations, each version of a type has full description in order to reduce the overhead of chasing the hierarchy to get the complete definition of corresponding version of a product type whenever a configuration is constructed. However, in the process of deriving versions, we can take advantage of the characteristics of version relationship that there is usually a little difference between successive versions of a type. In derivation of a new version from an existing version we can classify the contents of the version definition into four groups: Inherit, Drop, Addto, and Vchange. Inherit properties are inherited to the new version in the derivation process. Drop properties are eliminated from the new version. Properties which need to be added to the new version should be placed in AddTo. Existing properties which need to be changed to versions of constituents in the new version should be placed in Vchange. This IS-VERSION-OF hierarchy is also used in the description and derivation of registered configurations.

4. Versions Derivation and Propagation

4.1 Taxonomy of Schema Modifications

This work exploits the notion of schema evolution in which types are versioned [2, 3, 11, 14, 19, 20] for managing product evolution in agile manufacturing environments. Evolutional changes to the types are allowed in this model, revolutional changes in which data unloading and reloading are required are not allowed. If a revolutional change is made to a type definition, the instance objects which were created under other versions of the type might not be accessed under the new revolutionary changed version without data reorganization. Thus, only evolutional changes are allowed. The following schema modifications are corresponded to changes to product definitions, i.e., variations of products. In [3], a taxonomy of schema changes is defined. We refine the taxonomy and classify it into three groups: evolutional modifications, revolutional modifications, and pseudo revolutional modifications. The refined taxonomy is as follows.

1. Changes to the components of a type

1.1 Changes to attributes:

- 1.1.1 Add a new attribute
- 1.1.2 Drop an attribute
- 1.1.3 Change the name of an attribute
- 1.1.4 Change the type(domain) of an attribute
- 1.1.5 Change constraints
- 1.1.6 Change semantics
- 1.1.7 Change the default values

1.1.8 Change the inheritance(parent) of attribute(Inherit another attribute with the same name)

1.2 Changes to methods:

1.2.1 Add a new method

1.2.2 Drop a method

1.2.3 Change the name of a method

1.2.4 Change the code of a method

1.2.5 Change the inheritance(parent) of a method

2. Changes to inheritance graph or tree:

2.1 Add a new supertype or subtype relationship between types

2.2 Remove a supertype or subtype relationship between types

2.3 Change the inheritance ordering (multiple inheritance systems)

3. Changes to types themselves:

3.1 Add a new type

3.2 Drop an existing type

3.3 Change the name of a type

3.4 Merge types

3.5 Split types

Evolutional modifications: The changes of schema does not affect the instance objects which were created under previous versions of the schema. Despite the change of schema, we can access all the instance objects created under different versions of the schema without unloading and reloading instance objects. But a special facility should be provided to support it[6, 14, 19]. The followings can be included in this category.

1.1.1 Add a new attribute

1.1.2 Drop an attribute

1.1.5 Change constraints

1.1.7 Change the default values

1.2.1 Add a new method

1.2.2 Drop a method

1.2.4 Change the code of a method

2.1 Add a new supertype or subtype relationship between types

2.2 Remove a supertype or subtype relationship between types

3.1 Add a new type

3.2 Drop an existing type

Revolutional modifications: We can not accommodate this kind of modifications without unloading and reloading instance objects. We can not access instance objects which were created under different versions of such modifications. The followings are included in this category.

3.4 Merge types

3.5 Split types

Pseudo revolutional modifications: We can accommodate this kind of modifications without unloading and reloading instance objects if we provide a complicated facility for it. At the present time, since it is too complicate and is not cost-effective we consider these modifications as revolutional ones. The followings are included in this category.

1.1.3 Change the name of an attribute
1.1.4 Change the type(domain) of an attribute
1.1.6 Change semantics
1.1.8 Change the inheritance(parent) of attribute
1.2.3 Change the name of a method
1.2.5 Change the inheritance(parent) of a method

2.3 Change the inheritance ordering

In this work, only evolutional schema modifications are considered as legal schema changes since we deal with data evolution not revolution which is accompanied by unloading and reloading data instances. No other schema modifications are allowed.

4.2 Version Derivation

Versions of types correspond to variations of composite products which evolve constantly in agile manufacturing environments. We have to maintain version derivation history and other information in order to expedite version derivation and provide flexible configurations of products. An anchor type object[8] is used to keep version derivation history and version management information. When the first version of a type is derived, its anchor type object is created. The introduction of anchor type objects raises the problem of identifiers(OID); what is the identifier of the anchor type object? There are several methods solving this problem. We adopt the following mechanism. Let A be an type object and A_a and A_0 be an anchor type object and the original type definition which corresponds to initial design of a product respectively. The identifier of A_0 is assigned to the identifier of A_a and a new identifier is created for A_0. The objects referencing to A_0 are now pointing to A_a. In this way, we don't need to correct objects that are referencing to A_0. A flag saying an anchor type is placed in A_a to distinguish it from conventional versions of types. References to A_a should be resolved to a specific version of the type during the construction of product configurations. An anchor type object has the following information.

- Flag : represents an anchor type
- Type property : supertype, abstract, or concrete type
- Version count : the number of existing versions
- Version derivation hierarchy[8] : references to the version derivation hierarchy.
- The version derivation hierarchy is a tree of version indicators, one for each version of the type. A version indicator

has the following attributes.

- Version number,
- Timestamp : the creation time of the version,
- OID of the version definition,
- Version property
- Parent version number
- Child version number

Versions can be classified into frozen, stable, and transient versions. A version is frozen when any instance objects are created under the version and stored in the database. A frozen version cannot be deleted. A version is stable if it has descendent versions and it can not be deleted unless its descendent versions are deleted. A transient version is a leaf version in the version derivation hierarchy. It can be deleted and modified.

A version is pseudo-exclusive version if it is derived due to the changes to the interface attributes of its containing products. This version is owned by a specific version of its containing type which is derived due to the propagation. References to the anchor type can not be resolved to the pseudo-exclusive version in the construction of configurations.

4.3 Version Propagation

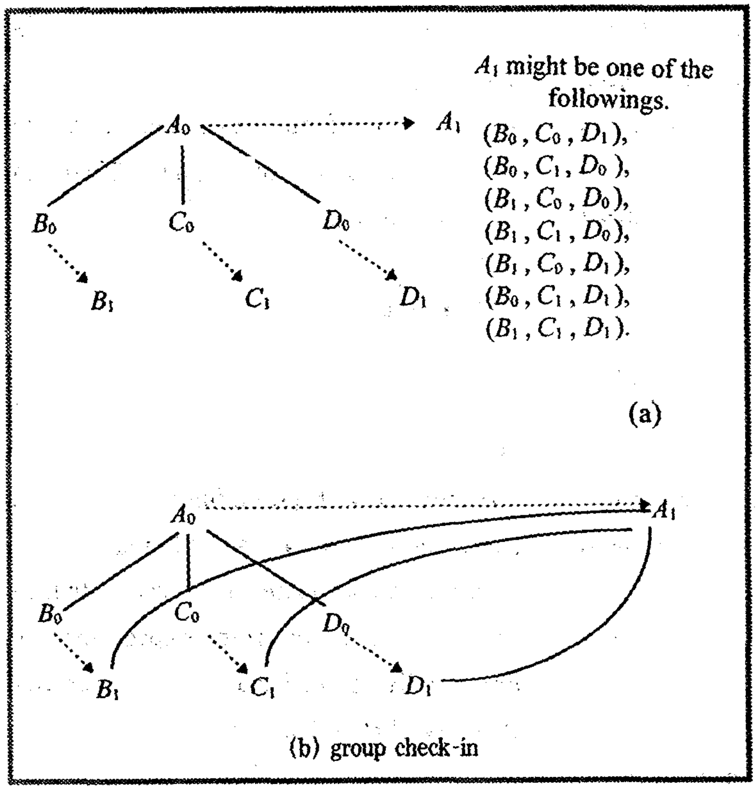
When a new version of a type is derived, it needs to be propagated along the hierarchies to reflect the changes and to keep consistency among versions of configurable parts. A simple solution to version propagation is to propagate all the versions along the hierarchy. This solution has a few drawbacks to be applied to agile manufacturing environments.

First, it leads to version proliferation and gives rise to the overhead of maintaining and manipulating versions.

Second, when more than one version is derived simultaneously, we have difficulty propagating these versions as we see in Figure 1. [10] proposed group check-in to solve this problem. However, group check-in is not a perfect solution. It gives rise to a problem that valid configurations might be lost in the propagation process. In our approach, all the versions are not propagated to the containing types. propagation occurs only when versions are derived due to the changes to the interface attributes.

Thus we avoid version proliferation, however we provide flexible configurations without losing of valid configurations. Version propagation algorithms have been developed.

〈Figure 2〉 Version propagation



Algorithm VerPropagation(Product, Version)
begin
If it is change to interface attributes of itself
　then for each containing product, OwnerVersion of OwnerProduct
　　if it is an interface attribute of the containing product
　　　then VerPropagation (OwnerProduct, OwnerVersion)
　　endif
endif
end

5. Configuration Management

5.1 Configurability Maps

Each version of a composite product can have several alternative versions for each component product. Assume that a composite product C has m component products, each has v versions. There are vm combinations for the versions of the composite product. Not all of the possible combinations are compatible among them. Hence consistency checking is needed for selecting valid configurations from possible combinations.

There are two approaches to handling the validity of configurations: configuration maps and configuration integrity rules. There are three methods to managing configuration maps: traditional data base approach, extended database approach, and functional approach. In traditional database method, all the possible valid combinations are represented explicitly. Every entry for the configurable map is an atomic value. Assume that the height of a composite product C is h, each version of all component products of C is comprised of c component products in average, and each product including C has v versions in average.

The total number of maps is calculated as follows.

$$v + cv + c^2v + \ldots + c^hv = \frac{v(c^h - 1)}{c - 1}$$

Each map has v^c entries at maximum. This method is straightforward, but gives rise to

the overhead of space and maintenance of the maps, especially when the number of version and component products is large. In extended database method, each entry of the map does not need to be an atomic value, it can be a set value or a range of version numbers since a set of, or a range of version numbers can be an object in extended database or object-oriented paradigm. Thus we can reduce the size of a configurability map. In functional method, a set of expressions is used instead of explicit version numbers of component types. In this method, we can represent configurability based on the properties of its component types. Configurability rules are specified in the rule-based approach. If we combine the functional approach with this approach by having configuration integrity rules reflect the functional relationship, we might expect benefits from this approach. There is a shortcoming with this approach. As the size of the rules increases, the performance is apt to degrade. There are two types of rules;

1. α,
2. $\alpha \Rightarrow \beta$

where α and β are expressions in which any number of predicates are connected by $\neg$, $\wedge$, $\vee$. A predicate consists of an attribute of a type definition, relational operators($=$, $<$, $>$, $\neq$, $\leq$, $\geq$) and constant values or an attribute of a type definition. Every symbol in rule expressions is universally quantified unless specified otherwise. Let us take some examples.

(PassengerCar. person ≥6) ∧ (PassengerCar. style = sedam)
PassengerCar. chassis. body. length ≥ 400.
Auto. chassis. body. length > Auto. chassis. body. width.

The performance of each method can be different depending on the situation; number of version, number of component types, complexity of validity, etc.. Extended database approach and rule-based approach can be promising candidates. In the implementation of this paper, we adopt extended database approach. If we represent each valid configuration explicitly, we get too many entries in a configurability map as we mentioned in the above. Additionally configurability map adjustment would be time consuming followed by the derivation of a new version of component products. We use the following structure to represent configurability maps. A configuration map for a version of a composite type consists of entry item and a set of version numbers for each component type. An entry item means a component type and its version, which was derived newly. There is a set of version numbers compatible with the entry item for every component type except its own type. All the elements of each set of component types compatible with the entry item are not directly specified in the configuration map; a set identifier is given in the map and the elements of the sets are stored separately to reduce duplication. Configurability map adjustment and configuration validity checking algorithms are given in the following respectively.

```
Algorithm MapAdjust(Product, Version):
// Product is a type and Version is a version of the type which is derived due to the changesto its interface attributes.
begin
  For each containing product, OwnerProduct:
  begin
   if OwnerProduct is not a supertype
     then begin
       For each version, OwnerVersion of the containing product, OwnerProduct
           InsertMap(Product, Version, OwnerProduct, OwnerVersion):
           end
     endif
end;

Algorithm InsertMap(Product, Version, OwnerProduct, OwnerVersion):
// Product and Version is derived type and version
// OwnerProduct and OwnerVersion is containing type and version
begin
  Put Version in the entry item:
  for each component of OwnerVersion except Product do
   begin
     Find compatible versions with Version, let it be CompatiSet:
```

```
      If a set that is the same as CompatiSet in values exists in the map
        then put the identifier of the existing set in corresponding position of the map
        else begin
          Store CompatiSet;
          Put the identifier of CompatiSet in the corresponding position of the map;
          end;
    endif
    end;
end;
```

```
Algorithm Function ValidityCheck (Product, Version, Components) : Boolean;
// This function receives a composite type with a version and a set of its component types with version numbers and
// return its validity value.
// Product is a composite type and Version is a version of Product.
// Components is a set of its component types with version numbers.
begin
  RemainingCompon := Components;
  NonEntry := empty;
  Valid := true;
  Find the latest element of RemainingCompon and let it be Late;
  // A version with the latest timestamp is chosen.
  while Valid and (RemainingCompon is not empty) do
    begin
    RemainingCompon := RemainingCompon - Late
    if Late exists in the entry item of the configurability map of Version;
      then begin
        Find the latest element of RemainingCompo and let it be CheckingOne;
        Get the corresponding place for the type of CheckingOne in the map;
        TempNon := NonEntry;
        While Valid and ( TempNon is not empty) do
          begin
           Choose one of TempNon and let it be NonOne;
          if there is the version of NonOne in the corresponding place
```

```
                    then TempNon := TempNon - NonOne
                     else Valid := false
               endif
           end
   if there is the version of CheckingOne in the corresponding place
       then Late := CheckingOne
       else Valid := false
   endif
   end
  else begin
       NonEntry := NonEntry + Late; //Add Late to the NonEntry set
    end
endif
   end;
   If Valid = true then ValidityCheck := true
                  else ValidityCheck := false;
   endif
end;
```

5.2 Registered Configurations

We choose a configuration among valid configurations of a version of a composite type in order to produce products. A configuration for a composite product is defined recursively. A configuration for a version of a composite type is defined by its component types and their version numbers. If a version of the component type is a composite type, it is required to have a configuration of it. Each configuration represents a variation of configurations. Hence, there are two kinds of forms for representing configurations. For the case the component type is a composite type, it is represented as a collection of (component type name, the version number of the component type, a configuration number of the version of the component type). For the case that the component type is not a composite type, it is represented as a collection of (component type name, the version number of the component type). We have several reasons to maintain product configurations. First, it might be called on later to produce the same product. Second, a new configuration might be derived from an existing configuration due to several reasons; modifying, fixing, or enhancing existing products. Third, each object instance should have configuration information since a version number can not distinguish instance objects for a composite product. Configuration numbers can be used in maintenance environments to support traceability. Hence we maintain configurations in the version-of hierarchy which was mentioned in the above. We need a special object called anchor configuration object to keep configuration derivation history like the anchor type object in version derivations of types. An anchor configuration object has the following information.

- Type and version number : represents the type and version number for which the configuration is defined.
- Configuration count : the number of existing configurations of the type and version
- Configuration derivation hierarchy : references to the configuration derivation hierarchy

The configuration derivation hierarchy is a tree of configuration indicators, one for each configuration of the given version of the type. A configuration indicator has the following information.

- Configuration number
- Timestamp : the creation time of the configuration
- OID of the configuration
- Configuration property; registered, stable, or transient
- Parent configuration numbers
- Child configuration numbers
- Creator of the configuration
- Characteristics : explains the characteristics of the configuration.

A configuration is registered if instance objects of the configuration are created. We need a few algorithms to manage configurations. Algorithms to create a new configuration from the product hierarchy and to derive a configuration from existing configurations have been developed. The algorithm creating a new configuration from the product hierarchy is given in the following.

```
Algorithm CreateConfig(Product, Version, Configuration);
// Create a configuration of a given version of a product respectively, Version, Product.
// Product is a composite and concrete type.
begin
Find all component types; put AND components in AQueue, for each OR component, determine whether it should be
included or not, if it should be, then put it in the queue;
for each component type in the queue do
  begin
    If it is XOR type // check if it is a super and abstract type.
      then begin
          choose one of its highest non-abstract subtypes and replace this with the XOR type;
          Determine version number;
          end
      endif
   end;
// Check the validity of the configuration using configurability maps.
if ValidityCheck(Product, Version, AQueue)
    then create a configuration of Version of Product with AQueue
    else exit giving the message "(Product, Version, Aqueue) is not valid configuration"
endif
// Now we are to determine configurations of component types which are composite types
for each component type, Component, with a version number, CompoVersion do
    If it is a composite type
       then begin
           Call CreateConfig(Component, CompoVersion, AConfig );
           If AConfig is new
             then Create a configuration of Aconfig for CompoVersion of Component;
           endif
           end
    endif
end;
```

6. Conclusion and Further Work

An object-oriented approach has been proposed to manage evolving composite-products in agile manufacturing environments. Schema evolution modification facilities are provided to support full potential versioning of product type definitions. Variants of the products are represented as versions of types. A well-defined representation scheme for composite products is introduced to provide systematic description of composite products and to be exploited in version control. All of the possible versions of a composite product are not explicitly represented to avoid version proliferation. Version derivations due to the changes to the interface attributes are propagated. However, valid configurations of any composite products can be provided to comply with customer demands. Traceability issues have been considered in the development of our approach even though they are not dealt with directly in this work. They will be included in the continuing further work. Configurability maps have been introduced to cope with configurability issues. Only valid configurations can be provided to the users through checking configurability using the maps.

There are several issues to be researched further. First, interfacing association relationship should be further investigated. Second, traceability issues need to be incorporated in the further work.

〈References〉

[1] Ahmed, Rafi and Navathe, Shamkant B., "Version Management of Composite Objects in CAD databases", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1991, pp. 218-227

[2] Anderson, L. et al, "Panel on Schema Evolution and Version Management", *SIGMOD Record*, Vol. 18, No. 3, September 1989.

[3] Banerjee, J., Kim, W., and Korth, H. F. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", *Proceedings of the ACM SIGMOD International Conference on Management of Data*. May 1987, pp. 311-322.

[4] Batory, D. and Kim, W., "Modeling concepts for VLSI CAD objects", *ACM Trans. Database Systems*, Vol. 10, No. 3, 1985, pp. 322-346

[5] Beech, D. and Mahbod, B. "Generalized Version Control in an Object-Oriented Databases", *Proceedings of*

IEEE 4th International Conference on Data Engineering, Feb. 1988, pp. 14-22.

[6] Cattell, R. G. G, *Object Data Management: Object-Oriented and Extended relational Database Systems.* Addison-Wesley, Reading MA, 1994

[7] Dittrich, K. and Lorie, R. "Version Support for Engineering Database Systems", *IEEE Trans. on Software Engineering*, Vol. 14, No. 4, April 1988, pp. 429-437.

[8] Chou, H. T. and Kim, W. "Versions and Change Notification in an Object-Oriented Database Systems", *Proceedings of 25th ACM/IEEE Design Automation Conference*, 1988, pp. 275-281.

[9] Jin, M. and Ting, T. C., Configuration Management of Evolving Products in Agile Manufacturing Environments, *Proceedings of the ISCA International Conference*, 1995, pp. 308-312

[10] Katz, Randy H., "Toward a Unified Framework for Version Modeling in Engineering Databases"" *ACM Computing Surveys*, Vol. 22, No. 4, Dec. 1990, pp. 375-407.

[11] Kim, W. and Chou, H. T., "Versions of Schema for Object-Oriented Databases", *Proceedings of the 14th VLDB Conference* 1988, pp. 148-159.

[12] Kim, W., Bertino, E., and Garza, J. F. "Composite Objects Revisited", *SIGMOD* 1989, pp. 337-347.

[13] Klahold, P., Schlageter, G., and Wilkes, W., "A General Model for Version Management in Databases", In *Proceedings of the 12th International Conference on VLDB*, 1986, pp. 319-327.

[14] Monk, S. R. and Sommerville, I., A Model for Versioning of Classes in Objet-Oriented Databases, *Proceedings of the 10th British National Conference on Databases*, July 1992, pp. 42-58.

[15] Nguyen, G. T. and Riew, D., "Schema change propagation in object-oriented databases", *Information Processing* 89, IFIP 89, pp. 815-820.

[16] Rumbaugh, J., "Controlling propagation of operations using attributes on relations", In *Proceedings of the OOPSLA*, 1988, pp. 285-296

[17] Sciore, Edward, "Multidimensional Versioning for Object-Oriented Databases", *Proceedings of DOOD* 1991, pp. 356-370.

[18] Sciore Edward, "Versioning and Configuration Management in an Object-Oriented Data Model", *VLDB Journal*, Vol. 3, No. 1, 1994, pp. 77-106

[19] Skarra, A. H. and Zdonik, S. B. "Type Evolution in an Object-Oriented Database", *Research Directions in Object-Oriented Systems*, 1987, pp. 393-413.

[20] Zicari, R. and Altair, GIP, "A Framework for Schema Updates in An Object-Oriented Database System", *Proceedings of the IEEE Conference on Data Engineering* 1991, pp. 2-13 or *Building an Object-Oriented Database System, The O2 story*, Morgan-Kaufmann, 1992.