Evolution of Schema and Individuals of Configurable Products

Tomi Männistö and Reijo Sulonen

Helsinki University of Technology
TAI Research Centre and Laboratory of Information Processing Science
Product Data Management Group
P.O. Box 9555, FIN-02015 HUT, Finland
Tomi.Mannisto@hut.fi, Reijo.Sulonen@hut.fi
http://www.cs.hut.fi/u/pdmg

Abstract. The increasing importance of better customisation of industrial products has led to development of configurable products. They allow companies to provide product families with a large number, typically millions of variants. Description and management of a large product variety within a single data model is challenging and requires a solid conceptual basis. This management differs from the schema evolution of traditional databases and the crucial distinction is the role of schema. For configurable products, the schema changes more frequently and more radically. In this paper, the characteristics that distinguish configurable products from traditional data modelling and management are investigated taking into account both the evolution of the schema and the instances. In this respect, the existing data modelling approaches and product data management systems are inadequate. Therefore, a new conceptual framework for product data management systems of configurable products that allows relatively independent evolution of schema and instances is proposed.

1 Introduction

To satisfy the demands of individual customers, companies need to provide a larger variety of products. Products, or *product families,* that allow large variation in a routine manner are called *configurable products.* The routine adaptation necessitates that the product family is pre-designed to cover a variety of situations. One challenge is to find the correct concepts for modelling configurable products so that the descriptions can be kept up to date as the product evolves. The challenge has not been adequately answered by the research in product configuration [1, 2], product modelling standards [3, 4] or commercial product data management (PDM) systems [see, e.g., 5].

In the following, we argue that a major reason for the inadequacy in the solutions is the problematic role of schema of configurable products. From the product modeller's viewpoint, modelling of configurable products involves the following levels [3, 4]:

- *The basic concepts* of the chosen data modelling method. For example, 'class', 'attribute', 'classification', 'inheritance' and so on.
- *The product modelling concepts* described using the basic concepts. Examples include 'product', 'has-part relation', 'optional part' and 'alternative parts' and conditions, such as 'incompatibility of components'.
- *Product family descriptions* formed using the product modelling concepts. These descriptions are also called *configuration models.*
- Finally, *product individuals* are instantiated according to the configuration model.

From ontological viewpoint, the two first correspond to *top level ontology* and *domain ontology,* respectively [6], and the two latter are an application of the domain ontology to a specific case. Data modelling, however, typically differs from the above view by modelling the *world* using the levels *concepts, schema* and *individuals.*

- *Concepts* are principal data modelling elements for articulation about the world.
- *Schema* is the actual model of the world; it defines the entity types of the world.
- *Individuals* constitute the actual population of data that can be queried and manipulated. Individuals in a database represent the individuals of the world that the schema describes.

Configuration modelling does not fit nicely into the data modelling view for various reasons. Firstly, configuration models need new concepts, such as 'has-part relation with alternative parts', not available as traditional data modelling concepts.

Secondly, as companies develop their products, product family descriptions are constantly changed. Product family descriptions, however, conceptually correspond to a traditional schema. In traditional database schema evolution, the existing data is typically converted to reflect the changes in the schema. In product development, this is not the case—old product individuals are not typically converted as product is developed. The evolution of a schema in product modelling, therefore, significantly differs from traditional schema evolution.

Thirdly, individuals of configurable products have long lifetimes and histories of their own. More importantly, they do evolve independently of the schema since customers may change their product individuals as they please. This is a crucial difference to traditional databases since the evolution of individuals is not reflected in the schema and consequently, the individuals do not necessarily conform to the schema. Modelling such individuals necessitates flexibility, for example, relaxing the strict conformance of individuals to the schema.

Therefore, in this paper we search for a suitable mechanism for modelling the evolution of configurable products. The mechanism should capture both the evolution of the schema and the individuals for an environment in which the conformance of individuals to the schema is not constantly maintained.

2 Previous Work

The basic goal in product configuration modelling is to provide means for describing a large set of product variants by a single data model. Methods of artificial intelligence have been extensively used for modelling the conditions that define the valid

combinations of components and for finding a solution for particular requirements [1, 2]. There are also approaches for representing product variety based on the product structure, such as generic Bills-of-Materials or generic product structures [see, e.g., 7], and approaches that emphasise product structure and classification in configuration modelling [8, 9]. Although the difficulties in managing configuration models are widely recognised, the vast majority of the models ignore all aspects related to the evolution of configuration models and configurations. In this paper, we aim at providing the concepts for capturing such evolution.

Version is a concept for describing the evolution of products [10, 11]. Versions typically represent the evolution of a *generic object* that, in turn, has a set of versions. A generic object is sometimes also called a *generic version* or *generic instance* [12, 13, 14, 15]. There can be two kinds of *component references* to versions: *statically bound* and *dynamically bound* [13]. A statically bound reference specifies explicitly a component and one of its versions. A dynamic component reference, i.e., a component reference to a generic object, can be bound to a specific version at various points in time. The idea is that in a given context, one of the versions from the set is a representative for the generic object, that is, the one to which dynamic references are bound. In the following, we utilise the version set approach, but for both the schema and the individuals in parallel.

Schema evolution is a relevant issue for databases. The approaches to schema evolution can be classified to *filtering, persistent screening* and *conversion* [16]. In filtering, a database manager needs to provide filters so that an operation defined on a particular *type version* can be applied to any instance of any version of the type [17]. Persistent screening defines how the instances should be modified to accommodate a schema modification, but the actual conversion takes place only when an instance is accessed for the first time [18]. Conversion, on the other hand, modifies all instances right after the schema modification [16].

A principal assumption in database schema evolution is that instances can be converted. In product configuration, however, this is not always possible. For example, product development may have obsoleted some components and thus removed them from a new configuration model. Therefore, the old product instances that were configured and manufactured according to an old configuration model are not represented by the new model. Thus, automatic conversion of old product individuals to the new schema is not meaningful. Because individuals cannot be converted, it is natural to have individuals from different versions of schema.

3 Evolution of Schema and Individuals

In this section, we position this work by identifying four basic cases according to whether the evolution of the schema and/or the individuals is supported. By supporting evolution, we mean here that the history or consequences of a change are supported in a more advanced form than just by allowing modifications without leaving any traces in which order they were done.

3.1 Category 1: No Support for Evolution

No support for evolution means that there is no memory or history of the schema or the individuals. Typically, the schema is assumed static and the individuals are mutated "in place". These mutations are typically required to be such that the individuals are always (outside transactions) consistent with respect to the schema.

As an example, assume 'X-Bike' with two models 'Standard X' and 'Deluxe X'. In the schema, 'Standard X' and 'Deluxe X' would be subtypes of 'X-Bike'. Typically, individuals would be created only as instances of 'Standard X' and 'Deluxe X', for example, 'Deluxe X, serial #1183'.

3.2 Category 2: No Schema Evolution, but Individuals Evolve

Although most databases do not have any concept of history, some databases store the history of individuals. With history stored, one can go back in time and see how things were at some point. For example, one can find the description of a product individual at the time it was delivered. With more advanced temporal querying capabilities information can be retrieved using temporal relations. Nevertheless, in this category the schema is assumed stable although the evolution of individuals is supported.

With respect to the 'X-Bike', individuals such as 'Deluxe X, serial #1183' would have multiple data representations. For example, 'as-manufactured version' of 'Deluxe X, serial #1183' would record the serial number of its critical parts, e.g., the frame, and customer specific options installed. Thereafter, all service operation are carefully recorded (this is a deluxe bike!). For example, as part of the regular service on March 15, 1999, the handlebar was changed to a new model 'Sporty W'.

3.3 Category 3: Schema Evolves, Individuals Do Not

Evolution of schema, when supported in a database system, typically propagates the modifications of the schema to instances as was discussed earlier.

For example in 1999, a new mountain bike model 'Bike XM' is introduced for 'Bike X'. The old models are still manufactured, although some components are replaced due to problems in durability and some due to the change of a subcontractor, both leading to new versions of 'Bike XM'. With some extra work, as-manufactured information can perhaps be found, but no explicit records of the individuals are kept.

3.4 Category 4: Both Schema and Individuals Evolve

This is the category most relevant to configuration modelling. This is actually what happens in the real world. The schema, i.e., the description of a product family, evolves as products are being developed. The developments in products cannot be propagated to the individuals automatically, as was discussed above. The schema evolves in this respect independently of the individuals. Customers, on the other hand,

can modify their product individuals practically as they please, that is, very much independently of the schema. Although, total freedom cannot be systematically supported, the relation of the schema and individuals should be maintained as far as possible.

For 'Bike X' this means that product families are modified and created, partly utilising the same components. The individual bikes are manufactured according to the product family descriptions, and the modifications to the individuals are recorded.

Many models lack the support for the evolution of the schema and individuals because of conceptual or practical simplifications rather than because it would not be needed. Especially in product data management, traceability and after-sales services need a history of the schema and individuals. If the evolution is not supported by the database management systems, the mechanism must be implemented on top of the existing system. The goal of this paper is to find solutions that could be implemented in PDM systems of configurable products. The important questions are "what does versioning of schema and individuals mean?" and "how do they relate?"

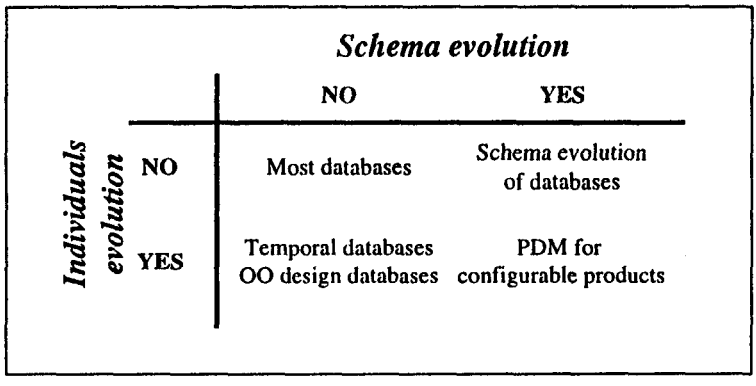

Fig. 1. Different approaches with respect to support for evolution of schema and individuals

In Fig. 1, the support for evolution of schema is shown on the horizontal axis, whereas the vertical axis shows the support for individuals. These axes generate a grid for categories 1–4, into which different approaches are placed.

4 Conceptual Model

It is impossible to discuss the evolution of schema and individuals unless there is some kind of agreement on their contents. Thus, we need to define some concepts. However, in order to keep a clear focus, there should be as few concepts as possible, and yet the concepts should capture the essential characteristics of configurable products. The objective of this paper is not to formally define the semantics of the concepts; the aim is more at providing an informal intuition of them. We build on our

previous work on product configuration [8, 9, 19] and, in this paper, add to that the evolutionary aspects.

We model both the schema and the individuals with the same concepts. The basic concepts are *object* and *relation* between objects. They provide a general and rather natural conceptual basis, which is also exemplified by their central role in many modelling formalisms, such as predicate logic, set and relation theory and so forth.

An object is either a *generic object* or a *version*. A generic object has a set of versions, and each version belongs to exactly one generic object. We use the term *type* to refer to an object in schema. A type is then either a *generic type* or a *type version*. Objects representing individuals, which we simply call *individuals*, are either *generic individuals* or *individual versions*. Note that the term 'object' refers to both the 'type' and the 'individual' and that we model the evolution of individuals by means of versioning. This means that all changes are implemented by creating new versions; versions themselves are not modified!

All relations can basically be reduced to relations between versions. For example, if A and B are generic objects, a relation between them is interpreted so that "each version of A is related to some version of B". This is the general idea, which we will elaborate further when discussing specific relations. Sometimes for simplicity, for a relation from A to B we say that A refers to B.

We use *effectivity* to organise the set of versions. Effectivity is the period the version was or is *effective* as a representative for the generic object. For modelling effectivity, we assume *discrete time* and relate each version with an *effectivity interval*. An effectivity interval consists of two time points, start time and end time. The end time is optional; an undefined end time meaning that the version is currently effective. The effectivity of a generic object is the union of effectivities of its versions. For simplicity, we also assume that at a particular time at most one version is effective for a generic object and that the effectivity intervals of consecutive versions meet. For resolving generic references, we search something more than simply using global time with effectivities. For example, we do not assume that an individual version effective at t is necessarily defined by a type version effective at t. Such situation results when product descriptions are modified because of product development but the product individuals are not converted. That is, the correct description, i.e., product type version, for an old product individual is not the new, effective one.

The association between individuals and types is modelled by two relations: *is-instance-of* and *conforms-to*. An individual is related to a type by is-instance-of relation. The "validity" of an individual with respect to a type is modelled by means of *conformance*, which is a condition stating whether the individual *conforms-to* the type it is related to by is-instance-of. With explicit is-instance-of, we want to stress that the type of an individual has been explicitly decided. Due to evolution, however, an individual may not be a valid representative of its type; this is modelled by conformance. These concepts are complex and powerful as such and the situation becomes even more complicated when generic objects and versions are introduced. Therefore, we try to give an intuition what we want to achieve with the concepts without unnecessarily going into too many details.

The concepts are roughly illustrated in Fig. 2, in which generic objects are represented by ovals. Inside an oval are the versions of the generic object as small circles, each with an effectivity interval next to it. Solid-line arrows represent is-instance-of relations and dotted-line arrows relations in schema and between individuals.

The concepts provide the basics for recording the history of objects. The concepts can be used in various ways, and therefore, the modelling of evolution is far from solved by just providing the basic concepts. Next, we enhance the model to support a meaningful evolution of schema and individuals together by defining informal invariants. The role of invariants is to constrain the underlying (imaginary) language and so approximate the intended situations [6]. A PDM system can then implement a selection of invariants to provide the wanted semantics.

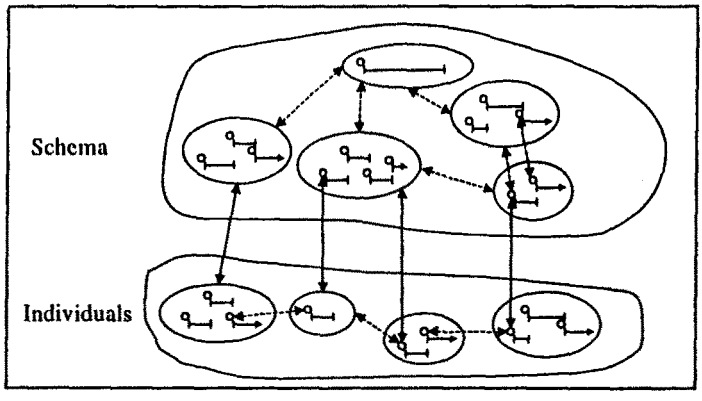

Fig. 2. Example of basic concepts in schema and individuals and relations between them

4.1 Is-instance-of Relation and Conformance

Is-instance-of is a relation between individuals (generic or versions) and types (generic or versions). For an individual, it denotes the type of the individual (if any). The validity of an individual with respect to its type is controlled by means of conformance as was discussed above. In particular, we are interested in is-instance-of relation with generic individual or type.

The basic intention is that each individual has a type and conforms to that type. More precisely, we mean conformance of an individual version to a particular type version although the is-instance-of relation can be between generic objects. We define two invariants, a strong and a weak, to articulate the semantics we want for configurable products. With the invariants, we also state to which type version the individual should conform in case of generic references.

Strong conformance invariant: *An individual is constantly kept in conformance to the type it is-instance-of.* (1)

The invariant 1 requires that at any given *t* an effective individual version is-instance-of an effective type version to which it also conforms. With respect to schema modifications, the invariant 1 covers the schema evolution of databases when individuals are converted immediately after a schema modification. Semantics for delayed conversion would be achieved if the conformance were required only for the creation time of each individual version. That would allow a generic individual to remain untouched regardless of schema changes. However, as soon as the generic individual is accessed, a new version that conforms to the effective type version is created. The semantics for filtering approaches, in which individual versions live according to their original type version, would be slightly different. For them, the is-instance-of is allowed only from a generic individual to the type version effective at the time the generic individual was created and each individual version is required to conform to that type version.

In all these approaches, each individual version conforms to some type version. For configuration modelling, however, this is too strict. For example, when a product individual is modified by changing components in it, the result is typically a mixture of original components and some new ones. Consequently, the modified product individual (version) does not necessarily conform to any configuration model. Therefore, we also define a weak conformance invariant.

Weak conformance invariant: *The first version of a generic individual conforms-to the type version it is-instance-of at the time of its creation.* (1')

A generic individual, i.e., its first version, is now created as an instance of a type, but may thereafter evolve independently of it. A conversion of a generic individual to a new type version is represented by creating a new, converted version and relating it by is-instance-of to the correct type version. (Note that the invariant 1 implies 1'.)

When a generic individual is-instance-of a generic type, it means that all versions of the generic individual are instances of some version of the type. If an individual version is-instance-of a type, this does not say anything about the consecutive versions of the generic individual. So, a generic individual can be used in is-instance-of relation to control the type of its future versions. This requires the creation of a new individual if the relation needs to be changed, for example, to a new type version.

4.2 Is-a Relation and Inheritance

Is-a is a relation in schema that serves mainly two purposes. First, it organises the types into a *classification taxonomy,* in which the "lower" objects are *specialisations,* also called *subtypes,* of the "higher" ones, also called *supertypes* of the former. This ordering bears the idea that the subtypes can be used in place of their supertypes. Second, is-a relation provides a mechanism for sharing common properties by means of inheritance from supertypes. We define invariants for controlling the effects that changes have via inheritance.

Strong effectivity invariant for is-a: *Effectivity of each (sub)type version must be contained in the effectivity of single version of its supertype* (2)

Consequently, a generic type may version without a need to update its supertypes, but for its subtypes, new versions need to be created. The invariant 2, therefore, guarantees that the inherited properties of a type version remain unchanged. This invariant provides a strict modelling basis in which the properties of a type version never change during its effectivity.

Existence of supertype invariant: *Effectivity of type must be contained in the effectivity of a (super)type it is-a (subtype of).* (2')

The invariant 2' is a weaker form of the invariant 2. The main difference between them is that the invariant 2' requires only that an effective version for the supertype exists, not that the changes in it are propagated downwards. Consequently, the inherited properties of a type version may change during its effectivity.

4.3 Has-part Relation

Has-part relation differs from is-a by occurring in schema and between individuals (but not between an object in schema and an individual). The detailed semantics of has-part [8] is not important here since we are interested in the evolutionary aspects. We define similar invariants as for is-a.

Strong effectivity invariant for has-part: *Effectivity of a version must be contained in the effectivity of a version it has as part.* (3)

Existence of part invariant: *Effectivity of an object must be contained in the effectivity of an object it has as part.* (3')

The invariants are written so that they can be applied to has-part between schema as well as between individuals. For has-part between types, the former corresponds to the versioning semantics in which a modification to a component is propagated to all wholes using it as part. The latter corresponds to the semantics in which components may version independently, typically as long as the changes are internal to the component.

Discussion on the use of generic individuals in has-part is similar to that of is-instance-of. A has-part relation from a generic individual states that all its future versions have the part, which may be too strong. For types, however, one may want to make such a statement thus requiring a creation of a new generic type, not only a new type version, in case the part needs to be changed. A has-part to a generic individual allows modification, e.g., servicing, of an individual component without the need to change the whole (in case the existence invariant 3' is used).

4.4 Combination of Is-instance-of, Is-a and Has-part

We have now defined three kinds of invariants: strong, existence and weak. The weak invariant was defined only for conformance since we wanted to support the evolution of individual rather independently of the types. We could also have defined a weak

invariant for is-a as well. That would be an invariant that requires only the creation time of a type to be contained in the effectivity of its supertype. Such invariant would allow evolution of type independently of its supertypes. We wanted such independence between individuals and types, but not for the type hierarchy, for which we want to control the evolution more strictly.

For a data management system, the important question is what forms of the invariants should be selected. For a traditional database, the strong ones are typically the most appropriate as has already been discussed. Our intention in this paper is to find a suitable selection for a PDM system of configurable products. A reasonably good choice of semantics for such system could use:

- weak conformance invariant for is-instance-of (i.e., 1'),
- strong effectivity invariant for is-a (i.e., 2) and
- existence of part invariant for has-part (i.e., 3').

This selection of semantics requires strict control on the changes to the classification hierarchy. That is, a change in a type necessitates its propagation to the subtypes and the properties of type version cannot change during its effectivity.

Has-part, however, reflects the evolution of components in a company. It is typical to allow certain modifications to a component, i.e., creation of new versions, without propagating them in the product structures. The allowed modifications are sometimes defined as those that maintain the "form, fit and function" of the component. Such semantics is achieved with existence invariant. For individuals, this also allows modification of a component individual without versioning the whole product individual.

For the relation of individuals to their types, we have already argued that changes to product definitions are not automatically propagated to the individuals and the individuals may be modified independently of their types. However, product individuals should be created according to a product type. This is captured by the weak conformance invariant, which requires the creation time conformance of the generic individual but allows independent evolution thereafter. This means that only the first version of a generic individual must conform to its type. If the generic individual is later converted to another type (version), this can be recorded by creating a new, converted individual version as is-instance-of the type. This, however, requires that one has defined is-instance-of relation for the first version of the generic individual, not for the generic individual itself.

5 Conclusions

Problems addressed in this paper are those of configurable products. The concepts presented, however, do not directly reflect the concepts of configurable products. The explanation is that the need for dealing with both the evolution of product types and the product individuals is characteristic of configurable products. In mass-products, the evolution of single product individual is not recorded. In very complex project products, there is no model from which the individuals could be instantiated. Configurable products are somewhere in the middle combining the routine aspects of mass-products, e.g., the pre-defined product types, and uniqueness of product individuals of

complex products, for which the evolution of individuals is more interesting. Configurable products thus provide a problem of their own for the evolution in data modelling systems. We provided a conceptual framework for managing such evolution.

There are other data modelling approaches that might support the needed evolution. One special approach is to abandon the distinction between classes and instances altogether [19]. This provides some degree of freedom for describing configurable products but cannot escape the problems of evolution [20]. Even with no classes and instances, there will be class-like objects and objects that represent individuals.

Schema evolution of databases, on the other hand, is based on the principle that schema constantly represents the same (or at least almost the same) real world entities. Consequently, the conversion of old instances is meaningful. For configurable products, this does not hold and therefore, we had to discard the strict conformance between the schema and individuals.

As most versioning approaches concentrate on modelling design objects, the methods have not been applied to configuration models. Typically versioning of design objects is represented at instance level and the evolution of schema, if present, is similar to the schema evolution of databases [15, 21, 22].

In product modelling, the largest initiative is the STEP standardisation effort [23]. STEP addressed the modelling of products for their whole lifetime. Modelling the evolution of product individual is well catered for, but there are problems in representing product families, not to mention their evolution [4].

It is hard to provide semantics that would suit for all PDM system of configurable products. Nevertheless, we provided a selection of invariants that covers the most typical of cases. The conceptual framework is derived from the experiences we have with two dozen companies that are manufacturing configurable products. Therefore, we dare to claim that although the feasibility of the approach presented has not been validated, it is not hypothetical.

Acknowledgements

We gratefully acknowledge the financial support from Technology Development Centre of Finland (TEKES) and Helsinki Graduate School of Computer Science and Engineering. We also thank Hannu Peltonen and Timo Soininen for their valuable comments on the paper.

References

1. Sabin D., Weigel R.: Product configuration Frameworks—A survey. IEEE Intelligent Systems & Their Applications 13:4 (1998) 42-49
2. Darr T., McGuinness D., Klein M.: Special Issue on Configuration Design. AI EDAM 12:4 (1998)

3. McKay A., Erens F., Bloor M.S.: Relating Product Definition and Product Variety. Research in Engineering Design 8:2 (1996) 63-80
4. Männistö T., Peltonen H., Martio A., Sulonen R.: Modelling generic product structures in STEP. Computer-Aided Design 30:14 (1998) 1111-1118
5. Miller E., MacKrell J., Mendel A: *PDM Buyer's Guide*. 6th edition CIMdata Corp. (1997)
6. Guarino N.: Formal Ontology and Information Systems. In: In Guarino N (ed.) Formal Ontology in Information Systems. Proc. of FOIS, Amsterdam, IOS Press (1998)
7. Erens F.. *The Synthesis of Variety—Developing Product Families*. PhD thesis, Eindhoven University of Technology. 1996.
8. Soininen T., Tiihonen J., Männistö T., Sulonen R.: Towards a General Ontology of Configuration. AI EDAM 12:4 (1998)
9. Männistö T., Peltonen H., Sulonen R.: View to Product Configuration Knowledge Modelling and Evolution. In: In Faltings B and Freuder EC (eds.) Configuration—Papers From the 1996 AAAI Fall Symposium (AAAI Tech. Report FS-96-03), AAAI (1996) 111-118
10. Katz R.H.: Toward a unified framework for version modeling in engineering databases. ACM Computing Surveys 22:4 (1990) 375-408
11. Dittrich K.R., Lorie R.A.: Version support for engineering database systems. IEEE Transactions on Software Engineering 14:4 (1988) 429-436
12. Katz R.H., Chang E., Bhateja R.: Version Modeling Concepts for Computer-Aided Design Databases. Proc. of SIGMOD (1986) 379-386
13. Kim W., Banerjee J., Chou H.T.: Composite Object Support in an Object-Oriented Database System. Proc. of OOPSLA (1987) 118-125
14. Biliris A.: Database Support for Evolving Design Objects. Proc. of the 26th ACM/IEEE Design Automation Conference (1989) 258-263
15. Batory D.S., Kim W.: Modeling concepts for VLSI CAD objects. ACM Transactions on Database Systems 10:3 (1985) 322-346
16. Penney J., Stein J.: Class Modification in the GemStone Object-Oriented DBMS. Proc. of OOPSLA (1987) 111-117
17. Skarra A.H., Zdonik S.B.: Type Evolution in an Object-Oriented Database. In: Shiver B and Wegner P (eds.) Directions in OO Programming, MIT Press (1988) 393-415
18. Banerjee J., Kim W., Kim H.-J., Korth H.F.: Semantics and Implementation of Schema Evolution in Object-Oriented Databases. Proc. of SIGMOD (1987) 311-322
19. Peltonen H., Männistö T., Alho K., Sulonen R.: Product Configurations—An Application for Prototype Object Approach. Proc. of the 8th European Conference on Object-Oriented Programming (ECOOP '94), Springer-Verlag (1994) 513-534
20. Demaid A., Zucker J.: Prototype-oriented representation of engineering design knowledge. Artificial Intelligence in Engineering 7 (1992) 47-61
21. Kim W., Bertino E., Garza J.F.: Composite Objects Revisited. Proc. of the International Conference on Management of Data (SIGMOD) (1989) 337-347
22. Nguyen G.T., Rieu D.: Schema evolution in object-oriented database systems. Data & Knowledge Engineering 4:1 (1989) 43-67
23. ISO Standard 10303-1: Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 1: Overview and Fundamental Principles (1994)