

OODBMS Metamodel Supporting Configuration Management of Large Applications¹

Piotr Habela¹ and Kazimerz Subieta²

^{1,2}Polish-Japanese Institute of Information Technology, Warsaw, Poland

²Institute of Computer Science PAS, Warsaw, Poland

Abstract. Many practical cases of database schema evolution require an effective support from configuration management. Although DBMS construction and software configuration management (SCM) constitute the well established areas of research, they are usually considered in separation from each other. In this paper different issues of SCM are summarized and their relevance to DBMS is investigated. We suggest to extend the OODBMS metamodel to allow recording certain aspects of application-database dependencies in a database schema repository. The extended metamodel contains both typical database metamodel information as well as software configuration information. Such a solution we consider necessary for solving some of schema evolution problems.

1 Introduction

The problem addressed in this paper emerges as an important aspect the definition of a metamodel for object-oriented databases. While the metamodel can be understood as just the description of the data model, in the context of a DBMS it is convenient to understand the term “database metamodel” as the description of all database properties that are independent on a particular database state. Using this view, we can identify the following roles the metamodel for a database system must fulfill:

- **Data Model Description.** The metamodel needs to specify the interdependencies among concepts used to build the model, some constraints, and abstract syntax of data description statements; thus it suggests the intended usage of the model.
- **Implementation of DBMS.** A metamodel determines the organization of a metabase. It is internally implemented in DBMS as a basis for database operations, including database administration, internal optimization, data access and security.
- **Generic programming.** The metamodel together with appropriate access functions become a part of the programmer’s interface for programming through reflection, similarly to Dynamic SQL or CORBA Dynamic Invocation Interface.
- **Schema evolution.** A metamodel equipped with data manipulation facilities on metadata supports schema evolution. Although it is relatively easy to provide the schema modification mechanism itself, the impact of such modifications presents a significant challenge to the configuration management solutions.

¹ This work is partly supported by the EU 5th Framework project ICONS, IST-2001-32429.

As can be seen from the above outline, the database metamodel definition has to address many different and to some extent contradictory requirements. The well known proposals in this area, especially the ODMG [2] standard whose metamodel definition we refer to, are rather far from fulfilling the mentioned roles. Since the mature and well known definition of an OO database metamodel is still not available, we will introduce and briefly describe our own sketch of such metamodel.

In this paper we discuss the consequences of addressing the last of mentioned roles by a database metamodel. This feature is considered to be of great importance for modern DBMS and is supported in a number of commercial products. Unfortunately, the ODMG standard touches this issue only implicitly and, as will be shown, for different reasons inadequately.

The problem of schema evolution remains an active area of research and resulted in many papers (e.g. [3,5,6,13,15,16,17]). The majority of these proposals, although inspiring, can be perceived as too idealistic for today's software development practice. Taking a more pragmatic approach, we would not deal with attempts to automatize the schema evolution process, but instead – propose features of database schema, intended to support Software Configuration Management (SCM). Assuming that in most cases there is only one valid schema version used by a system, our aim is to provide means to easily and reliably extract the information needed to adjust the database applications according to intended schema change.

The paper is organized as follows. Section 2 summarizes the issues of schema evolution, explains why it is inseparable from the SCM area, and presents the requirements of configuration management, identifying the different kinds of configuration information and their relations to the DBMS schema. Section 3 outlines the core elements of proposed object-oriented database metamodel, and identifies possibilities to store configuration information within it. Section 4 concludes and outlines the key issues that need to be considered in the future research.

2 Schema evolution and SCM

Schema evolution has been recognized as an inevitable aspect of long lived information systems maintenance. The DBMS mechanisms intended to support it are considered mainly in the context of object-oriented databases and became one of the most prominent features to be introduced [1].

Although the research on this subject resulted in more than hundred papers, the problem is far from being solved. Naive approaches reduce the problem to operations on the metadata repository. This is a minor problem, which can be simply solved (with no research) by removing an old schema and inserting a new schema from scratch. If database application software is designed according to SCM principles, then the documentation concerning an old and a new schema must be stored in the SCM repository. Hence, storing historical information on previous database schemata in a metadata repository (as postulated by some papers) in majority of cases is useless.

The features supporting schema evolution of OO databases has not been effectively standardized so far. The ODMG standard only implicitly assumes such functionality. The interfaces used to define its metamodel provide the modification operations, and

their presence is adequate only in the context of schema evolution. However, as already stated, the schema evolution problem cannot be reduced to more or less sophisticated operations on the schema alone. After changes in a database schema the corresponding database objects must be reorganized to satisfy the typing constraints induced by the new schema. Moreover, application programs acting on the database must be altered. For this reasons, serious treatment of SCM excludes ad hoc, undocumented changes in the database schema.

To highlight the real schema evolution problem we present the following real-life example from one of our database applications:

- Altering the schema (in SQL): **10 minutes**
- Preparation and performing of database conversion: ca. **2 days**
- Examination of some 400 potentially affected user screens, altering some of them, updating documentation and system testing: **several person-months**.

This example makes it possible to realize that the schema evolution capabilities in the ODMG standard address only the initial hours of many months of work.

2.1 Views, wrappers and mediators

Some papers devoted to schema evolution assume that the problem can be solved by database views. After changing a schema one can define views, which provide virtual mappings from the existing objects to the new schema; hence no changes occur in database object and no changes in existing applications is required. Alternatively, one can convert objects according to the new schema and define views, which preserve the old schema for already defined applications. In both cases, old applications need not be altered, hence the major problem of schema evolution is solved.

In the majority of cases such an approach is idealistic for the following reasons:

- Some changes in a schema stem from unsatisfactory properties of applications, hence changes of applications are inevitable.
- Some changes in a schema stem from changes in business data ontology (e.g. implied by new law regulations). Any automatic mapping of existing data is unable to fulfill new business requirements.
- View definition languages are not sufficiently powerful to cover all possible mappings. There are many mappings not covered by SQL views.
- The view updating problem is solved only in specific cases and (probably) will never be solved in the general case. Hence many applications that require advanced view updating cannot rely on this approach.
- Access to data through views may result in unacceptable degradation in performance. Although materialized views partly solve the problem, this approach implies disadvantages: an additional storage and updating overhead.
- Applications written in languages such as C++ and Java are tightly coupled to physical properties of database objects. Such dependencies (sometimes undocumented and low-level) limit the use of database views.

Another approach to schema evolution can be based on concepts such as wrappers and mediators. The approach is similar to the approach employing database views, but in contrast to database views, which are defined in high-level query languages (SQL), wrappers and mediators are proprietary solutions, tightly coupled to a category of

applications and written in lower-level languages (C/C++, Java, etc.). The approach is more realistic than the approach based on database views, but it requires low-level programming. Some of the above mentioned disadvantages of using views are also true for the approach based on wrappers and mediators. In particular, if a change concerns data representation or business data ontology then any kind of wrapper or mediator may be unable to isolate the problem: the change would affect applications.

In summary, although database views provide some hope for schema evolution, according to our experience, this approach is non-applicable in majority of cases. More detailed discussion on this topic can be found in [19].

2.2 Schema evolution and software change management

Looking at the problem from the software engineering perspective, schema evolution forms a part of a more general topic, which is referred to as *software change management*. It concerns the maintenance phase in the software life cycle. The cost of maintenance is very high and in total can several times exceed the cost of initial software development. Thus some discipline is necessary to reduce the cost. Software change management provides activities during the software development, operation and maintenance to support software changes. It also determines disciplined processes for altering software after changes. Both of these aspects are important. If software developers and clients neglect certain activities addressing future software changes, then the cost of changes can be extremely high. Changes to the software should follow some life cycle to reduce cost and time, and to achieve proper software quality.

Considering schema evolution as a part of the entire change management process we see many change management activities that must be carried out in order to ensure proper conditions for schema evolution. The software change management must also define activities to establish an organized change process, in particular, the following:

- The procedures of reporting software problems or requesting functionality changes.
- Collecting and storing software problem reports; organizing their assessment and qualification according to importance, urgency, cost and the impact of the change.
- The diagnosis of software problems and cost estimations of software changes.
- Decision processes concerning the scope of software changes and/or making new versions of the software.
- Planning, organizing and scheduling concerning the software changes implementation, testing and documentation.
- Testing changed software according to software testing plan including regression testing (testing unchanged modules that can be influenced by the change).
- Documenting of changes, including requirements and other documentation.
- Installation of changed software, training of users and acceptance tests.
- Learning from change, to improve the change processes in the future.

A proposal concerning schema evolution should refer to the activities presented above in software development, to determine a clear goal for the research. It can be formulated in terms of cost, time or quality of particular activities, and/or in terms of software quality. The schema evolution capabilities like that defined in the ODMG standard, have no relationship to activities of the change management processes.

2.3 SCM repository and a metabase repository

SCM is a discipline for establishing and maintaining the integrity of the products of a software project throughout the project's lifecycle [8,9,10,11]. SCM is especially important if a project lasts several years and/or has many versions due to changing user requirements or system requirements. Schema evolution means a new version of a schema and, in consequence, a new version of the database, and a new version of applications. Thus, it must be disciplined by SCM.

A basic entity that SCM deals with is a software configuration item (SCI). An SCI can be atomic or complex. Complex SCIs include all software artifacts that present intermediate or final software products, including source code, documentation, tools, tests, etc. The SCM has to guarantee consistency among SCIs. SCI frozen for changes are called baselines. Some SCIs are called versions, revisions and releases.

All entities that are used or produced within a particular software version must be collected together as SCIs and stored within an SCM repository. This helps to avoid situations where new code is associated with old documentation; old code cannot be re-compiled because a relevant older compiler version is no longer available, etc.

As follows from the above, all versions of a schema must be the subject of SCM. The new schema must be stored within a consistent configuration which includes new requirements, diagnosis and analytical documentation, data conversion code, code of new application modules, new design and implementation documentation, testing code, data and results, software transfer documentation, user documentation, etc. Schema evolution cannot be separated from other SCM aspects and activities.

It is implicitly assumed in the research devoted to schema evolution (in particular, in the ODMG standard) that actions on the database schema repository will immediately change a repository state. Sometimes, it is assumed that the repository will also be prepared to keep historical information on previous schemata. Taking into account software change management and SCM, such an approach is inadequate. A change to a database schema must be carried out on the SCM repository, which should be prepared to keep both old and new schemata. Many other documents are related to this change, including software problem reports, new requirements, managerial decisions, software code, its documentation etc. All of this information must be kept within an SCM repository rather than within a metabase repository.

2.4 Dependencies between software units

Some tasks in software change management (like problem diagnosis, change planning and scheduling, implementation, testing, documentation updating etc). are more efficient (in terms of cost, time and quality) if the information on dependencies between software units could be properly organized.

Some dependencies between software units are or can be stored within a metabase repository. Other dependencies can be stored within an SCM repository, in particular, as SCIs. Below we list more important dependencies.

Configuration dependency: some software and documentation units are dependent because they create a consistent SCI. This dependency is usually stored within a configuration management repository. It is more relevant to SCM.

Forward dependency between procedural units of the software. The dependency shows which procedural units are called from a given procedural unit. This dependency is easy to discover by analysis of the code.

Backward dependency is exactly reverse to the forward dependency. It is more valuable than the previous one because it shows which software units calls a given unit. Both forward and backward dependencies are relevant to a metabase.

Event dependency holds between a unit raising an event and a unit catching it and triggering some action. The case is similar to forward and backward dependency. This information is usually present in the specification of interfaces (CORBA IDL, ODMG ODL), thus it can be stored within a metabase repository.

Parametric dependency between a given unit and a unit that can be a parameter to that unit. This concerns e.g. *call-by-reference* parameters of methods or parameters of some (generic) software templates. Parametric dependency is relevant to a metabase.

Side effects dependency describes all aspects of the data/computer environment that can be affected by a given procedural software unit. Side effects concern operations on a database, global data (shared among applications), hardware devices, catalogs, files, external communication etc. In languages such as Modula-2 and DBPL some side effects are explicitly determined by special programming facilities called import lists. Current object-oriented languages do not determine side effects within class interfaces, hence the programmer and the system is unable to recognize them directly. This can be the source of serious bugs, cf. the Ariane-5 rocket disaster caused by an unspecified side effect. Side effects can be passive (a read-only access), or active (affecting the state of some external resources). Providing the information on those dependencies is an obligatory part of a software unit specification. A metabase repository can store information on those side effects that concern a database. For instance, a part of database can be read and updated by a given method.

Definitional dependency holds between two data units, where one is a definition and another one is an instance of this definition. The dependency concerns interfaces, classes, types, patterns, skeletons, templates, schemas, specifications, etc. and their instances. Definitional dependency is relevant to a metabase and SCM.

Redundancy dependency holds between units that contain redundant information; e.g. copies of some data that are introduced to improve performance or to increase safety. Redundancy dependency is relevant to a metabase and SCM.

Taking into account the entire population of software and documentation units we can imagine their structure as a (partly directed) colored graph, where each edge represents some dependency between units and the color of an edge represents a dependency kind. Some dependencies in this graph form subsets of software/documentation units, in particular, configuration and definitional dependency. Some dependencies can be stored within a metabase repository. Other dependencies are more relevant to a software configuration repository.

In summary, the properly defined schema evolution problem should establish dependencies among software and documentation units. It should clearly subdivide the dependencies between a metabase repository and a configuration repository, and should clearly determine benefits of storing the information on dependencies for particular phases and aspects of the software life cycle, including the software change management. None of the above mentioned aspects of schema evolution are taken into account in the metamodel defined by the ODMG standard.

Concluding, the schema evolution problem far exceeds the pure problem of metadata management and should be considered as a part of software change and SCM. While some repository updating operations would indeed be useful, e.g. adding a new attribute or adding a new class, the operations do not solve the essential problem. The major problem of schema evolution concerns altering a database and – most of all – altering applications that operate on the database. This problem is related to software engineering rather than to the pure database domain.

3 Database metamodel support for configuration management

The main challenge of today's software development is dealing with complexity. In case of SCM this concerns especially the complexity of interdependencies among configuration items. Therefore, in order to better support the SCM aspect, the database metamodel definition should provide means to simplify the management of dependency information. There seem to be two general ways towards this objective:

- **Encapsulation / layered architecture.** Applying the encapsulation, both to narrow the interface of particular classes, as well as to isolate the whole layers, allows to shorten the dependency paths.
- **Dependency tracking.** Even if the dependencies don't span across many configuration items, they still need to be recorded in a way that would guarantee completeness of such information and ability to easily extract it.

Both of these postulates are rather intuitive and are presently treated as a *sine qua non* in the development of large information systems. However, the dependencies between applications and database schema constitute a special kind of dependency that would be much more effectively handled when supported by the core DBMS mechanisms. As certain kinds of dependency information concern directly the database schema elements, storing them within the schema would not significantly complicate the metadata structure.² It would be especially advantageous in presence of a DBMS mechanism that would enforce the dependency recording.

3.1 Dependency kinds relevant to the metabase

In the subsections we revisit those of earlier enumerated dependency kinds, we considered relevant to the metabase, looking for optimum way of storing such information within a DBMS schema.

Forward, backward and event dependency

Since these kinds of dependency are mutually symmetrical, they could be registered using single construct, e.g. bi-directional association. Assuming traditional architecture, we would be interested in dependencies between external applications

² Making the database aware of its dependent applications has previously been suggested e.g. in [4] through the concept of "application table". Intention of introducing that construct was slightly different though.

and the database, as well as dependencies within the DBMS (the DBMS dependencies of other system elements, as the least critical, would not be tracked).

The target of the dependency association would be any (that is, behavioral or static) property of the database. The role of a dependent element would be played by either DBMS native procedure / method, or by an external application's procedure or module. Therefore, in addition to the regular database schema elements and dependency association, we need to introduce construct, identifying an external procedure that the schema would be otherwise not aware of.

The optimum level of granularity of such information should be determined. The dependent element would be always a procedure / method. However, in case of external applications' dependencies, it could be practical to use a higher level, e.g. a whole application module. The target of a dependency can be either an interface or – assuming more detailed tracking – those of its properties that a given routine accesses.

Concerning the event dependency, it is also desirable to store information on both directions of that dependency that is on both the event producer and event consumer.

Side effect dependency

All requests to properties that are non-local for a given procedural unit or interface, can be qualified as side effect dependency. It is desirable to distinguish between passive and active side effects and to include this information in the metabase.

However, it is necessary to note that when we separate interface definition from the structure storing its instances, both the whole metamodel as well as the dependency tracking features, get more complex. The assumption, that the entity definition is inseparable from the set of its instances, is characteristic for the traditional relational model and contributes to its simplicity. We are not going to follow this approach though. The concept of *extent*, being a single structure containing all instances of a given class or interface, becomes problematic in case of distributed environment or (especially) when arbitrarily complex composite objects are supported.

Thus, it is not enough to connect the side effect dependency information with particular interfaces. The side effect dependency record should identify the global property the manipulated properties are accessed through. For example we would like to know not only that a given procedure refers to objects of type *Product*, but also that it operates e.g. only on objects stored within the global variable *availableProductsCatalog*. Therefore, in order to describe the side effect dependency it is necessary to identify the global variable a given procedure uses to begin its navigation, as well as all properties it then uses to retrieve the target reference. In case of static properties, each such dependency would be marked as read-only or updating.

Parametric dependency

This kind of dependency seems to be easier to handle than the side effect dependency, because here the dependent procedure does not have to be aware of the origin of provided parameter. No matter what kind of parameter it is: either the procedure reads, updates or returns newly created instance, it is only necessary to guarantee, that the definition of a given type stored in database metamodel has not been changed in a way that affect that procedure. In this case the target of dependency link would be simply a type definition the parameter refers to.

3.2 Proposed metamodel extensions

Before illustrating suggested features for dependency tracking, we will briefly describe suggested improvements to the conceptual view of the ODMG's metamodel. For a detailed discussion of the database metamodel roles and the requirements they entail, see [7].

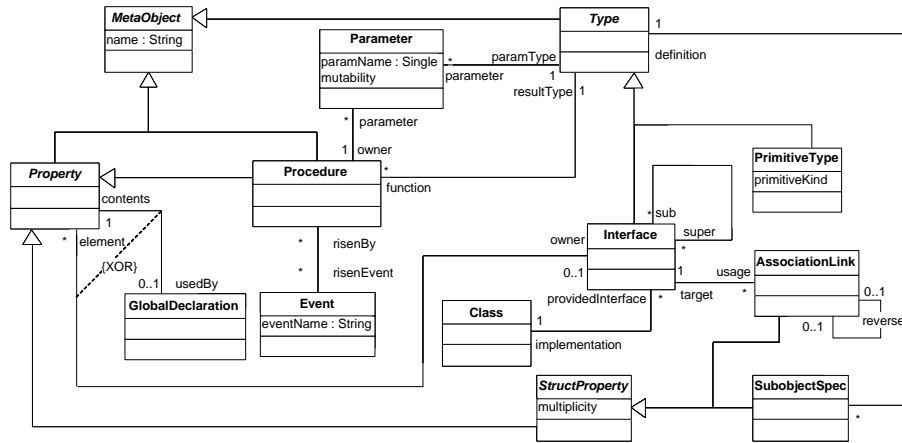


Fig. 1. Core elements of proposed metamodel without features specific to SCM support

Core elements of the suggested metamodel

The interrelations among the introduced concepts (Fig. 1) are presented in a form similar to the UML standard definition [14]; also, the graphical notation of that language is used. Below outline suggested properties of the improved metamodel.

Clear separation between meta-levels. Metamodel does not deal with concepts from other meta-levels, like e.g. *Object*. However, since the database schema has to be aware of the location of user objects and because we are not going to solve this through the *extent* concept, we need to store the declarations of database global variables. If a *StructuralProperty* is connected with a *GlobalDeclaration* metaobject, it means that the former is not a part of Interface definition,³ but instead it constitutes a global variable declaration (e.g. an entry points for further navigation through the database). Nevertheless, despite the separation of meta-levels, we still assume that both user objects and metaobjects would be accessed in a uniform way.⁴

Lack of the explicit collection concept. It is often necessary to store multiple instances of some type, without necessarily wrapping them into another object. Thus, instead of introducing the concept of collection, the *multiplicity* meta-attribute of a structural property allows to declare it as multivalued (and/or optional).

³ Similarly, a *Procedure* can be declared global instead of being a method of a given Interface.

⁴ As far as possible we tend to rely on generic query language mechanisms when manipulating the metadata. Note the lack of metadata operations in the presented diagram (Fig. 1).

Object relativism and composite objects. It is desirable to reduce as far as possible the difference in handling of composite and primitive objects. As an important change in comparison to traditional OO programming languages we suggest to allow for arbitrary (perhaps multilevel) nesting of composite objects. Such composition can be declared using the *SubobjectLink* metaobjects.

Metamodel extensions for dependency information management

Since all constructs needed to describe the targets of different dependencies are already part of the metamodel, adapting it to store the dependency information requires only few additions (see Fig. 2).

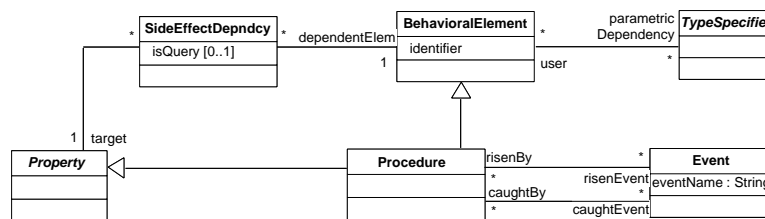


Fig. 2. The dependency management constructs as an extension to the presented metamodel

Because we are also interested in dependencies of elements located outside the DBMS responsibility (external applications using the database), a *BehavioralElement* concept has been added as a generalization of the *Procedure*. Both elements can be the source of dependency relationship: the former can be the external application's elements, while the latter denotes the native procedures stored within the DBMS.

The side effect dependencies are recorded for all elements that are used in navigation or manipulated. For each such dependency a *SideEffectDependency* metaobject connects the dependent element description with dependency target. The *isQuery* meta-attribute determines the character of dependency: either pure read / navigational (value *yes*) or allowing to modify a given element (value *no*). Note that it is not necessary to record the exact path of navigation. It is enough to indicate, whether any part of a given procedure refers to a given property or not. The *isQuery* value is applicable only when the dependency target is a structural property. Parametric dependencies descriptions refer to type definitions (that is, to primitive types or interfaces). Other relevant dependencies (e.g. event dependency and definitional dependency) also can be derived from the outlined metamodel structure.

In Fig. 3 we present exemplary fragment of schema, showing the side-effect dependency of an external procedure *updatePrices*. That method refers to a global variable *OfferedProducts* that is capable of storing arbitrary number of objects described by the interface *Product*. Through this interface, it can modify the subobject (attribute) *Price* of type *Currency*. In contrast, the global variable *OfferedProducts* is never modified by this procedure (it is used only for navigation).

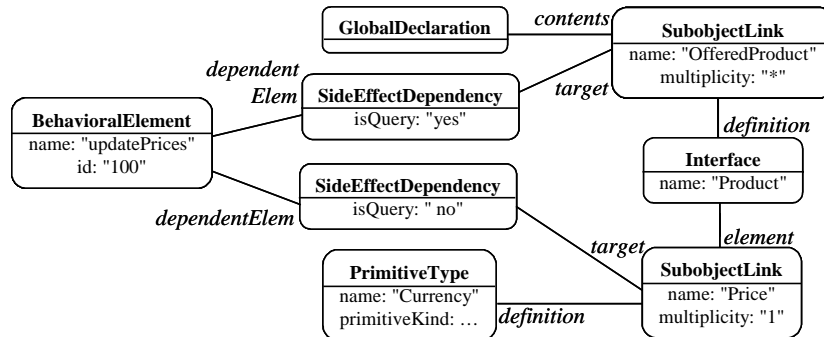


Fig. 3. An exemplary fragment of the DBMS schema, containing the side effect dependency information for an external procedure *updatePrices*

Collecting the dependency information

In recording the dependency information we have to face the problem similar to the one encountered between SCM system and other tools used in the software development: the development tools are not prepared for providing the information needed for documenting the software configuration. Inability to automatically extract all the dependency information makes it necessary to manually document it, which can be considered as much less efficient and reliable.

Therefore, to make the described feature practically useful, the ability to automatically collect the dependency information seems to be indispensable. In our case, the DBMS could record the dependencies during one phase of the system testing and verify the completeness of collected information during further tests. For each recorded dependency it would need to receive some id of the procedure or unit that performs the database call (e.g. in the form of additional parameter in the statement beginning the given interaction with the database).

Such identification process should be separated from regular database security mechanism in order to not to deteriorate the performance during regular operation.

4 Conclusions

The database schema definition constitutes the critical element of the configuration information that needs to be maintained during the system's lifecycle. Although it is usually relatively stable, it can be also a subject of change. The most significant potential impact of such change concerns the integrity of applications dependent on the schema. This aspect of the schema evolution issue, partly belonging to the SCM area, seems to receive a relatively little attention.

We argue that the database schema would be an appropriate place to store that kind of the dependency information. Therefore the metamodel for ODBMS should provide the necessary constructs. The key advantage in comparison to storing that information

in a SCM repository, would be the ability to record it automatically. This would require special mode of work of the DBMS, as well as the means to identify the external programs dependent on to particular properties of the database schema.

The discussion presented in this paper concern core DBMS mechanisms rather than a kind of external extension of existing systems. For this reason we discuss the necessary constructs in the context of new proposals in the object-oriented database area. Other important new features that such DBMS would benefit from include dynamic object roles [12], more powerful view mechanism [19] and the query language seamlessly integrated with the regular programming language constructs [18]. We are currently working on the database metamodel proposal, consistently incorporating the mentioned elements.

References

1. J.Banerjee, H. Chou, J.Garza, W.Kim, D.Woelk, N.Ballou. Data Model Issues for Object-Oriented Applications. ACM TOIS, April 1987.
2. R.Cattell, D.Barry. (eds.) The Object Data Standard: ODMG 3.0. Morgan Kaufmann, 2000.
3. K.T.Claypool, J.Jin, E.A.Rundensteiner. OQL SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. In Centre for Advanced Studies Conference, 1998, 108-122
4. F.Ferrandina, S.-E.Lautemann. An Integrated Approach to Schema Evolution for Object Databases. OOIS 1996, 280-294
5. E.Franconi, F.Grandi, F.Mandreoli. A Semantic Approach for Schema Evolution and Versioning in Object-Oriented Databases. Computational Logic 2000, 1048-1062
6. I.A.Goralwalla, D.Szafron, M.T.Özsu, R.J.Peters. A Temporal Approach to Managing Schema Evolution in Object Database Systems. DKE 28(1), 1998, 73-105
7. P.Habela, M.Roantree, K.Subieta. Flattening the Metamodel for Object Databases. (*To appear in*) ADBIS 2002.
8. IEEE Guide to Software Configuration Management, ANSI/IEEE Std 1042-1987
9. IEEE Standard Glossary of Software Engineering Technology, ANSI/IEEE Std 610.12-1990
10. IEEE Standard for Software Configuration Management Plans. ANSI/IEEE Std 828-1990
11. ISO/IEC 12207. Information Technology - Software Life Cycle Processes. ISO/IEC Copyright Office, Geneva, Switzerland, 1995
12. A.Jodlowski, P.Habela, J.Plodzien, K.Subieta. Dynamic Object Roles in Conceptual Modeling and Databases. Institute of Computer Science PAS Report 932, Warsaw, Dec. 2001 (submitted for publication).
13. S.-E.Lautemann. Change Management with Roles. DASFAA, 1999, 291-300
14. Object Management Group: Unified Modeling Language (UML) Specification. Version 1.4, September 2001 [<http://www.omg.org/>]
15. R.J.Peters, M.T.Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. TODS 22(1), 1997 75-114
16. Y.-G.Ra, E.A.Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Evolution. ICDE, 1995, 165-172
17. B.Staudt Lerner. A model for compound type changes encountered in schema evolution. ACM TODS 25(1), 2000, 83-127
18. K.Subieta. Object-Oriented Standards. Can ODMG OQL Be Extended to a Programming Language? (In) Cooperative Databases and Applications. World Scientific, 1997, 459-468.
19. K.Subieta. Mapping Heterogenous Ontologies through Object Views. Proc. of 3-rd Workshop Engineering Federated Information Systems (EFIS 2000), 2000, 1-10