

Figure 1: The different integration aspects within the IDEE architecture

Commercial product data management (PDM) systems are often advertised as the panacea for most of the above mentioned problems. But the current generation of PDM systems only manages the containers (files) in which the data are stored, instead of the data itself.

In the project IDEE (Integration of Data in Engineering Environments), we regard all the heterogeneous applications as local data management systems (although some of the applications are purely file-based). The goal of IDEE is to integrate these systems into a tightly coupled database federation using an object-oriented data model. Versioning of design data (at the object level) on the global layer and system-wide consistency are the main requirements imposed by our engineering domain. Other aspects covered by the project are the inclusion of global views and queries on the distributed data, workflow management, and the coupling of versioning and workflow with an appropriate access control concept (see figure 1).

To ensure the applicability of our approach in a productive environment, this project is executed in cooperation with an industrial partner, the electromechanical company ABB (Asea Brown Boveri). The IDEE system aims at integrating typical engineering applications, which are, e.g., in use in the gas turbine development and production departments of ABB in Baden, Switzerland. In a first step, IDEE will focus on the integration of data from design calculation programs, geometrical CAD data, and corresponding bill-of-material (BOM) data.

The remainder of this paper presents an overview of the IDEE project, problem analysis, and first design decisions. In section 2, we show the IDEE architecture, including integration alternatives for engineering applications (subsection 2.3). Section 3 discusses our choice of ODMG-93 as the global data model (subsection 3.1), explains usability specifications for object identity (subsection 3.2), and presents the global versioning approach (subsection 3.3). The paper concludes with our plans for future research in section 4.

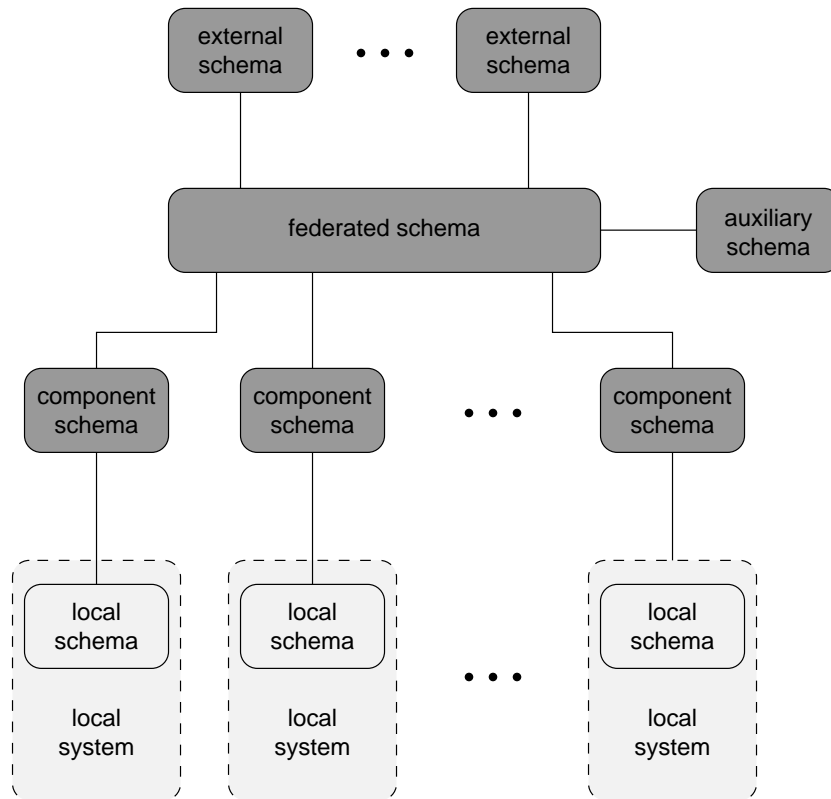


Figure 2: Four level schema architecture for IDEE

## 2 The IDEE Architecture

### 2.1 Schema Architecture

The IDEE four level schema architecture (see figure 2) has been derived from the reference schema architecture for tightly coupled federated database management systems proposed by Sheth and Larson [SL90]. Loose coupling [HM85] is not applicable, mainly because we need to support a centralised global view in IDEE.

In our architecture, different *external schemas* provide specialised views on parts of the *federated schema*. We introduce a single federated schema to logically centralise the data, thus facilitating the enforcement of global integrity constraints. Global constraints ensure the consistency of engineering designs across several local systems. The *auxiliary schema* comprises the descriptions of data not available in any of the local systems, global constraints, and federation management information.

In the domain of IDEE, we often have to deal with applications which are used for experimental designs, i.e. their local schemas change almost as often as their data. It is neither manageable nor necessary to propagate each of these local schema changes to the component schema of the federation. Therefore, a component schema represents, in terms of the global data model, the parts of the local schema the component is willing to share in the federation. Thus, our component schema is a combination of the originally proposed component and export schemas of [SL90].

### 2.2 Operational Integration

When running the federated system, global applications working on the federated schema try to access local systems, while updates in the components are propagated through the integration layer. To translate operations between the schema levels, mappings must be defined. Two

approaches are distinguished in [BNPS89]: structural and operational mapping. When using *structural mapping* (sometimes also called *schema mapping*), correspondences between elements of two schemas (and some of their operations) are defined. But in some local systems of our engineering domain, the semantics of local data is completely hidden in the local applications and schema information is not available. Therefore, and because we are faced with highly heterogeneous local systems, structural mapping is not applicable. Hence, we use the operational mapping approach.

*Operational mapping* does not define schema, but only operation correspondences, thereby implying the use of object-oriented techniques: A common coupling interface encapsulates the data in the different local systems. Each individual local implementation inherits and refines this coupling interface, thus exploiting two important advantages: The integration process is clearly defined (implementation of the interface operations) and integration work is reduced due to the reuse of inherited design and code.

### 2.3 Coupling Local Systems

Different existing applications offer different ways to access their data. Since we want to preserve local autonomy as much as possible, existing interfaces will be used to couple local systems to the integration layer whenever applicable. So far, five different groups of local systems have been identified, based on how the coupling modules may access local data (see figure 3).

- *Systems using a database management system (DBMS)*

Coupling a CAx tool that uses a commercially available DBMS is done by (operational) mapping from the component schema to the local database schema.

- *Applications with an external programming interface*

Such an interface offers commands which are equivalent to the user interface controls, as well as operations which access internal structures of user data.

- *Applications with an internal storage module interface*

A storage module interface provides the opportunity to access low-level application data (e.g., lines or 3D objects of a CAD program). Application objects and operations of higher abstraction (construction of assemblies, calculation of the centre of gravity of a complex part) are not accessible.

- *Shared data files*

If a local system does not offer any means to access data via the application itself, the data files of the application can be used. This raises the problem of synchronising access to the files.

- *File exchange*

In case that the application's data file format is unknown or subject to frequent changes, the remaining possibility is to exchange files of some known (preferably standardised) exchange format. Compared to shared data files, the synchronisation problem is even worse in this case. Hence, data in exchange files will be read-only at the global layer.

In order to ensure global consistency within an engineering environment, we have to monitor the updates that are autonomously carried out within local applications. Monitors report changes to the global layer which cares for their proper propagation to sibling systems. Hence, the control flow is not strictly top-down, but also bottom-up. Several implementations of monitoring facilities are possible:

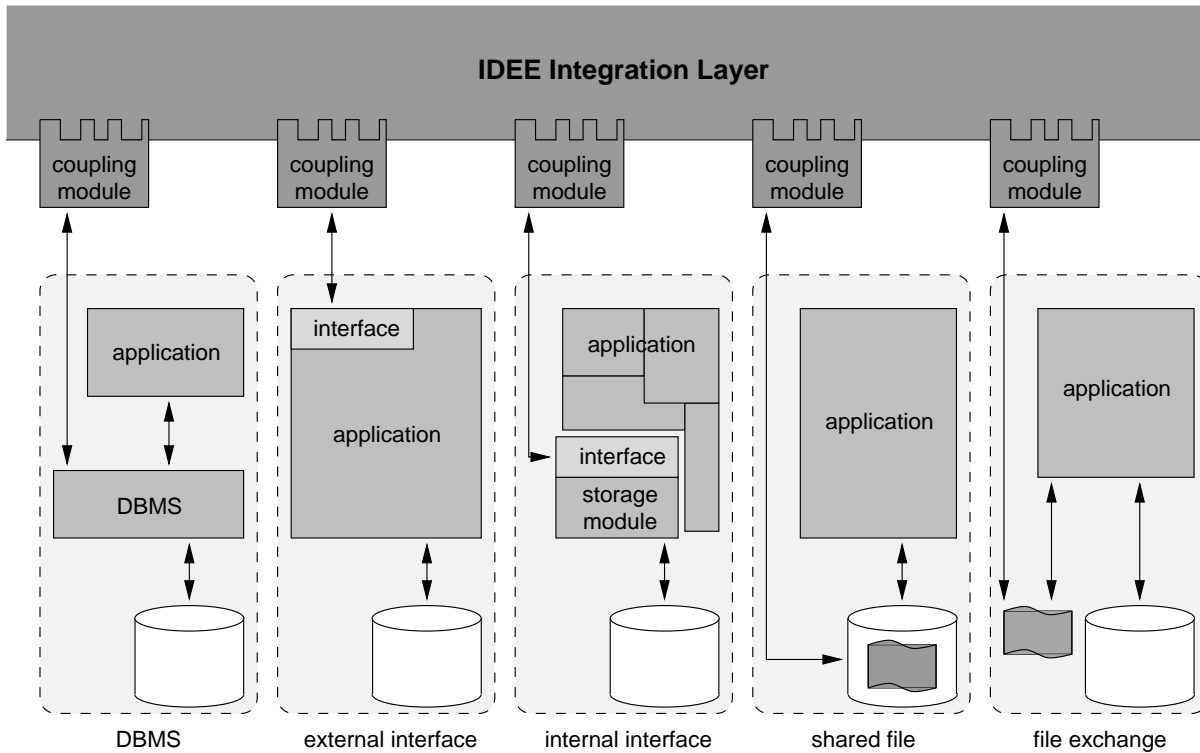


Figure 3: Interfaces to couple component systems

1. The local system may explicitly generate (e.g., through triggers) notifications or events and send them to the integration layer.
2. A local agent (e.g., a wrapper) can monitor the target application and inform the integration layer about updates.
3. Trail files or log files produced by the local application can be observed to collect information about recent changes.
4. If the local application delivers no information at all, regular calculations of file checksums are at least a way to discover that some data have changed locally. The exact changes have to be determined separately.

The implementations can be further subdivided depending on whether accesses of local applications to their local data are directly caught, or whether alerters are used which are attached to the data.

### 3 The Global Data Model

In IDEE, like in many other tightly coupled federations (e.g., COMANDOS [BNPS89], ZOO<sub>IFI</sub> [Här94, HD92]), an object-oriented global data model is used as this kind of model is regarded to be most suitable for the task. Their features [ABD<sup>+</sup>89] provide a rich expressiveness (i.e. the degree to which a real-world concept can be directly represented), which allows for the integration of almost any less or evenly expressive data model [SCG91]. While all IDEE-required features for global data models are covered, a few useful ones are missing in object-oriented models, for example, support for multiple semantics would be helpful to solve integration conflicts [SCG91]. Among the features of object-orientation, encapsulation is especially important for IDEE since operational integration is based on the encapsulation of heterogeneous component systems under a common object-oriented interface [BNPS89].

In this section, we explain why we have chosen ODMG–93 [Cat95] as the global data model of IDEE. Afterwards, we point out how the realisation of two aspects of the data model, namely global object identity and global version management, is heavily influenced by the application domain of a federation for engineering data.

### 3.1 ODMG–93 vs. STEP

Initially, we considered the following kinds of possible data models for IDEE:

- *A new IDEE-specific data model* would obviously be perfectly suited to our needs. But since it is unlikely to find support for it in industry, we judge the necessary efforts for design and implementation not worth the possible benefits.
- *Data models used in other research projects and prototypes* were not chosen for the same reason, namely the lack of industrial support.
- *Commercially available proprietary data models* (as used in some object-oriented databases, for example) were excluded, because we did not want to be stuck with some vendor.
- *A standardised data model* therefore had to be chosen, although industrial standards lack the latest research results, in favour of a rather pragmatic approach. However, standards increase portability of both data and applications, because they are supported by many different vendors. It is therefore necessary for the industry to enforce standardisation, and reasonable for IDEE to follow standards, too.

Since our project applies federated database technology to engineering data and applications, both, database and engineering standards were considered. The choice was reduced to the most prominent representatives of either side: ODMG–93 [Cat95], the database standard of the Object Data Management Group, and the ISO STandard for the Exchange of Product data, STEP [ISO94]. For IDEE, the advantages of ODMG–93 vs. STEP turned out to be overwhelming.

**The case for STEP** The strengths of STEP can be found in the specification of constraints and especially in the availability of standardised schemas (called “application protocols”) for many different application domains, e.g., general technical CAD drawings [ISO94, Part 201, explicit draughting] or car design [ISO94, Part 214, core data for automotive mechanical design processes]. If all local systems in a federation used (parts of) a standardised STEP schema as their component schema, the same schema could be used as the federated schema. Thus, neither schema nor semantic heterogeneity needed to be resolved during schema integration [BLN86]. However, our area of gas turbine development is not yet covered by a STEP protocol — we would depend on our own non-standardised schema.

Applications with a STEP interface are still rare in industry. However, the extension of any non-STEP systems with a STEP interface is “at least theoretically possible” [Dig92]. For an environment where all local systems have a STEP interface, the IDEE architecture (figure 1) could obviously be simplified, because a single coupling module would be sufficient for all local systems. But with ODMG–93 as the global data model, a similar simplification is possible: Local systems with an existing STEP interface can be integrated through a single coupling module which translates STEP schemas into their ODMG–93 equivalents [Lüh96].

**The case for ODMG–93** Advantages of ODMG–93 over STEP exist especially on the modelling side. Due to its heritage as an exchange file format, STEP can only represent static structures, it is only *structurally object-oriented* [Dit86] and therefore provides only *structural*

*expressiveness* [SCG91]. ODMG-93 additionally allows for the definition of methods and therefore behaviour — it is *behaviourally object-oriented* [Dit86] and has *behavioural expressiveness* [SCG91] (which, of course, is advantageous for integration).

Related problems with STEP as the global data model are due to its poor database support: It has no query language, no transactions, and is not well suited to be used by global users and applications. Apparently for these reasons, no STEP-based database management system is available. Hence, a DBMS with another data model would be needed to store the data of the auxiliary schema. To provide global applications with a more database-like interface of the federation layer, the other data model had to be layered above STEP. The necessary mapping between STEP and the other data model had to implement database concepts like queries and transactions on top of STEP.

ODMG-93 on the other hand, has been equipped with enough functionality to meet the needs of database application developers. Besides the common features of an object-oriented data model (data and object types, interface definitions with attributes, relationships, and operations, inheritance, etc.) the ODMG-93 object model also comprises several database-specific aspects like key and transaction semantics. ODMG-93 also defines the object query language OQL. Several (unfortunately only partially) ODMG-93 compliant database management systems exist on the market. However, not everything is perfect with the still evolving ODMG standard. For example, OQL may be suited for database experts, but rather not for engineers. A simpler interface for ad hoc queries is certainly necessary.

## 3.2 Object Identity

In object-oriented database management systems, each object has a unique identity which is independent of its state, immutable during its existence, and never reused [KC86]. But in object-oriented database federations, global object identity can only be partially provided without compromising local autonomy [EK91, Här94].

Two approaches to tackle this problem are rather obvious: Either the requirements on global object identity are generally weakened (as done in COMANDOS [BNPS89]) or local autonomy is reduced. The variety of systems we face in our application domain makes it unacceptable to take only one of these approaches into account. On one hand, weakening object identity to the least common denominator for all possible local systems would lead to the total loss of object identity, because some local systems do not support identity at all. On the other hand, trying to reduce local autonomy excludes local systems from integration for which autonomy cannot be reduced.

Hence, we will adapt an approach which was introduced in the *ZOO<sub>IFI</sub>* project [Här94, HD92]: the concept of *usability specifications*. This concept incorporates three levels of object identity which represent integration alternatives for global classes. For each global class, the specified level of global object identity determines which operations are applicable on the class' instances and which ones are not. The integrator must choose the appropriate level of identity for each class, based on its local representation(s). The global user perceives the chosen integration alternative as restrictions on the database functionality of global objects.

The strongest level of object identity in *ZOO<sub>IFI</sub>* is *permanent* identity, which does not impose any restriction on global objects. Objects are required to be always identifiable in their local system and to participate in global serialisation<sup>2</sup>. A typical candidate for permanent identification is, for instance, a global object which is based on a tuple in a relational database, given the tuple's primary key cannot be changed. Such a tuple can always be identified via the primary key and "locked" using the local transaction mechanism.

---

<sup>2</sup>This is usually implemented by means of some kind of local "locking" during global transactions. Such "locks" can be literal file system locks, but the use of semaphores or local transactions is possible as well.

A weaker form of identification, *temporary* identity, is valid only during a single (global) transaction (this concept is the one implemented in COMANDOS; it is also called *session oid* [EK91]). As an example, take a global object which represents one record from a structured file. If the file consists of records without keys, the object can be temporarily identified through its position in the file. Only if the file is “locked” by the global transaction, the object can be read from the local system (usually as the result of a global query) and even be updated. But the object may not participate in a global relationship, because the local system is free to reorganise the file as soon as the global transaction releases its “lock”. Since there is no key, it is impossible to identify the same record afterwards.

Finally, an *imaginary* identity is used for objects whose global representation cannot be related to the local representation, once the objects have been loaded. Examples for this weakest level of identification are data streams, and, similar to our example above, structured files for which no local “locking” mechanism is available. In this case, the local system may rearrange records immediately after the file has been read. Obviously, such objects are read-only.

The *ZOO<sub>IFI</sub>* project was focussed on the homogenisation problem in database federations. For the purpose of IDEE, the concept of usability specifications must be applied to ODMG-93 (*ZOO<sub>IFI</sub>* defined its own object model) and to the other aspects of our project, especially the version management.

### 3.3 Version Management

Using and changing versions of a particular engineering object (e.g., a product or a component) is the very nature of a design activity. Objects are reused and should evolve during the engineering process. Therefore, the global system must be able to identify and trace changes of the engineering object during its lifetime.

#### 3.3.1 Requirements

In its context of engineering design, version management in IDEE has to consider the following requirements:

- A global user wants to see a version of a product (e.g., a gas turbine) as a whole in any phase of the life cycle. References to parts of the product (e.g., the turbine blades) must be transparently resolved, i.e. the system must “know” which version of which part in which local system belongs to which version(s) of the product.
- Access to different versions of an object (in queries, for example) is necessary. Changes from one version to the next should be traced and a history of versions kept (who did what and when).
- Version management controls how objects are shared between different users. It should be possible to distinguish different modes of access to a specific version, such as read-only or read/write. Furthermore, access to a version should be restricted according to the current state of a design. A version should be kept *private* to its designer as long as it is incomplete. *Group* access can be granted when the design becomes usable for others who work on the same project. When the design is released (usually to be actually manufactured), it is made *public* and may be used as a part in other projects, too. These object sharing mechanisms are commonly known as *workspaces* (e.g., [Kat90]).
- In engineering design, the version history of an object can usually be visualised as a tree. This leads to the often useful distinction between *versions* and *revisions*<sup>3</sup>. A version is an alternative to its predecessor (thus creating a new branch in the version tree), while a revision is an improvement or correction (see figure 4).

---

<sup>3</sup>Subsequently, “versions” subsume “revisions” as long as their distinction is not necessary.

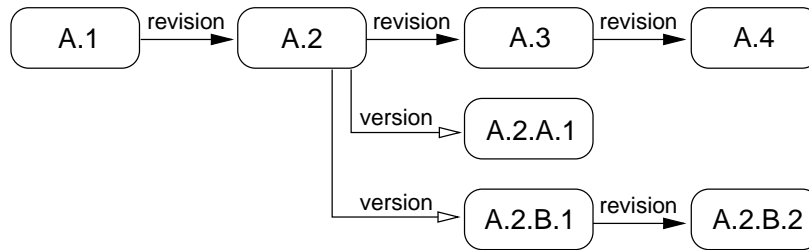


Figure 4: Versions and revisions

- Since constraints often change along with the designs they are imposed on, constraints must be versionable as well. This allows a constant evolution of the constraints by enforcing them only for data newer than a specified version. A similar argument holds for tools or calculation routines which operate on small sets of input data, but produce very large outputs which are better not stored permanently. These outputs are only reproducible if the correct version of the tool or routine is attached to its input data.

Numerous data models with integrated version management have been proposed and implemented, many of them especially developed for the needs of engineering databases (e.g., [Kat90, KBG89, DL88]). Since the global data model of our choice, ODMG-93, does not support versioning<sup>4</sup>, we will model a versioning mechanism by means of existing ODMG-93 constructs.

### 3.3.2 Coping with Different Local Systems

Global version management in a federated system has to cope with the different capabilities and restrictions of the different local systems. From the discussion in section 3.2 we can derive that only objects with permanent identification can be part of a global version history. Since a global version history implies global relationships between objects, temporary or even imaginary identified objects cannot be part of it<sup>5</sup>.

We can generally subdivide local systems into three groups according to their support for the implementation of global versioning on their data:

- *Local systems which provide an interface to their local version management that comprises the minimal functionality required to couple the global versioning scheme*

Such applications can be integrated into global version management without any restriction. However, since many different version and revision numbering schemes are used in the industrial practice, it is required to provide an adoptable coupling between the global and different local versioning schemes.

- *Local systems which only partially provide the required functionality*

Usually, these applications allow only support of a linear version history, i.e. only revisions. Global objects which represent data from these applications can therefore only be versioned with the same restriction.

- *Local systems which do not provide any versioning mechanism, which can be used by the global layer*

---

<sup>4</sup>Object versioning will not be included in the new standard ODMG-97 either, but has been deferred until a later release [ODM97].

<sup>5</sup>In future, we may consider to let a local versioning mechanism maintain a part of a “logically global” version history. It is then possible to keep global versions of certain temporary identified objects, e.g., objects based on relational tuples whose primary keys may change, but for whom referential integrity is nevertheless guaranteed by the local relational database management system.

This group also contains applications which offer version management for a local user, but do not provide (enough) insight into their mechanism, let alone an interface. In this case, the federation can access only one version (the last or *current* version [Kat90]) and no difference to a non-versioned application exists from the global point of view.

If no local versioning scheme is available for federation use, we must follow either a “local” or a “global” approach to cope with this situation, i.e. either a local versioning mechanism is added to the local system, or versions are managed and stored on the global layer. Both approaches offer several variants which have to be evaluated carefully and individually for each local system. We discuss their merits and deficiencies next.

**Local approach** There are several ways to extend a local system with a versioning mechanism:

- Where applicable, the most simple way is to introduce a version management tool to be used separately from local applications (like the GNU RCS for file-based systems, for instance). A slightly more advanced solution is the extension of local applications with an interface for the versioning tool, such that local users perceive the tool as a part of the application. The versioning tool can be employed to perform local versioning by both, the global system and local users.
- Alternatively, a local versioning extension can be provided for the federation’s use only, i.e. as an integration utility of the federation. Such a utility versions local data on the local system solely for the demands of the global versioning mechanism and invisible for local users.

These variants of the local approach have the advantage that not only data which is exported through the export schema can be versioned, but data sets that completely describe certain parts. This feature becomes essential if the exported data is itself not sufficient to run a local application, but represents only a fraction or even a summary of the required information. An example are the stress calculations for a turbine blade which depend on many different parameters. Most of them have a small impact on the calculations and are used for (mostly experimental) “fine-tuning” only. The important parameters are exported into the federation, because they represent the main characteristics of the design. To revert to an old version, all parameters of this version must be restored, otherwise the main parameters would be combined with the wrong, unrestored “fine-tuning” data. It is therefore necessary to save all relevant parameters each time a new version is created.

**Global approach** The global approach stores all but the current version on the global layer. Each time a new version is created by a global user, the current data are read from the local system and stored. When a global user wishes to go back to the old version, its data are written back to the local system and replace the current version. Note that this approach requires the implementation of transparent access to (old) versions of an object on the global layer and to the current version which is stored.

From the example above we know that this global approach usually requires to put all version-relevant information into the export schema. This leads to very large schemas and makes the approach not very appealing. But under the following conditions, it shows its benefits:

- If a local system allows us to update existing data but not to store newly created (version) information, we simply cannot use the local approach. Our choice is restricted to either dealing with large schemas or not using global versions.
- If things are even worse and we cannot write to a local system at all, the global approach allows us to keep at least a history of revisions (and only revisions, since we cannot create

branches in the version tree by going back to an old version). In this case, it is not necessary to export all data needed to restore a version. Hence the export schemas need not be extended. The price to be paid is the loss of intermediate results, which might become useful at some time, but cannot be reproduced.

- The global approach is also well suited if we do not *want* to write versions back to local systems. It can be an acceptable tradeoff to be satisfied with a read-only version history, if an object represents data originating from several local systems which do not support local versioning. The benefits for adding local mechanisms may simply be not worth the work.
- The global approach combined with a local addition allows to fully restore old versions without the massive extension of export schemas. The exported part of the version is stored on the global layer, as usual. Additionally, restoration information is exported which allows to restore the “fine-tuning” data (from local log-files or savepoints, for example).

**Consequences** Most consequences of either choice will finally be transparent to the global user, who does not care whether the versions he uses are stored on the federation layer or in local systems. But, comparable to the usability specifications for object identity in section 3.2, the global objects’ abilities to be versioned can be restricted through the capabilities of local systems they are based upon. The versioning mechanism in IDEE must therefore distinguish objects which are fully versionable, only “revisionable”, or not versionable at all.

## 4 Future Work

The next steps of IDEE comprise the detailed design of the integration layer, especially the versioning mechanism based on ODMG-93. A demonstrator will be implemented to validate the design.

In the long run, support for integrity constraints, transaction management, view and query facilities as well as workflow management and access control features can be added to the federation layer of IDEE. A generic integration framework is planned to extend the applicability of IDEE to other application domains.

## Acknowledgements

We thank Dirk Jonscher for his support during the course of the IDEE project, including this paper.

## References

- [ABD<sup>+</sup>89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Meier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International DOOD Conference, Kyoto, Japan, 1989*.
- [BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database system integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.
- [BNPS89] E. Bertino, M. Negri, G. Pelagatti, and L. Sbatella. Integration of heterogeneous database applications through an object-oriented interface. *Information Systems*, 14(5):407–420, 1989.

- [Cat95] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93 Release 1.2*. Morgan Kaufmann, 1995.
- [Dig92] Digital Equipment Corporation. Product data sharing using STEP technologies, February 1992.
- [Dit86] K. R. Dittrich. Object-oriented database systems: The notions and the issues. In *Proceedings of the International Workshop on Object-Oriented Database Systems, Asilomar, California, USA*, pages 2–4, 1986.
- [DL88] K. R. Dittrich and R. A. Lorie. Version support for engineering database systems. *IEEE Transactions on Software Engineering*, 14(4):429–437, April 1988.
- [EK91] F. Eliassen and R. Karlsen. Interoperability and object identity. *SIGMOD Record*, 20(4):25–29, December 1991.
- [Här94] M. Härtig. *Objektorientierte Integration von autonomen Datenhaltungssystemen*. PhD thesis, University of Zurich, 1994.
- [HD92] M. Härtig and K. R. Dittrich. An object-oriented integration framework for building heterogeneous database systems. In *Proceedings of the IFIP DS-5 Conference on Semantics in Interoperable Database Systems, Lorne, Australia*, pages 33–53, November 1992.
- [HM85] A. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.
- [ISO94] ISO IS 10303–1. STEP part 1: Overview and fundamental principles, Geneva, 1994.
- [Kat90] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.
- [KBG89] W. Kim, E. Bertino, and J. F. Garza. Composite object revisited. *SIGMOD Record*, 18(2):337–347, June 1989.
- [KC86] S. N. Khoshafian and G. P. Copeland. Object identity. In *Proceedings of the 1st OOPSLA Conference, Portland, Oregon*, pages 406–416, September 1986.
- [Lüh96] H. Lühsen. *Die Entwicklung von Datenbanken für das Produktmodell der ISO-Norm STEP*. PhD thesis, University of Erlangen–Nuremberg, 1996.
- [ODM97] ODMG. Draft Chapter 2 on the Object Model for Release 2.0. Available as <http://www.odmg.org/odmg93/updates/chap2x.ps>, April 1997.
- [SCG91] F. Saltor, M. G. Castellanos, and M. García-Solaco. Suitability of data models as canonical models for federated databases. *SIGMOD Record*, 20(4):44–48, December 1991.
- [SL90] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.