Semantic Modeling of Successive Events Applied to Version Management

J.H. ter Bekke
Department of Information Systems
Delft University of Technology
Zuidplantsoen 4, 2628 BZ Delft, The Netherlands
E-mail: j.h.terbekke@is.twi.tudelft.nl

Abstract

The semantic abstractions classification, aggregation and generalization are extremely useful for modeling complex situations containing time-independent events. This paper shows that they also can be used successfully for modeling of successive events. Here, two situations can be considered: ordering and sequencing. Both will be illustrated with practical examples on workflow and version/configuration management. The solutions do not require special constructs and can therefore be implemented in any programming environment.

Keywords*: data modeling, semantic databases, ordering, sequencing, version modeling, workflow management, design data management.*

1 Introduction

The semantic abstractions aggregation and generalization appeared for the first time in the database literature in a series of papers by Smith and Smith [11, 12, 13]. These abstractions are considered to be suitable for modeling complex situations in which different types of relationships occur. Because of their nature, they are especially useful for modeling hierarchical relationships. Many examples can be found in literature. Considering these application areas, only minor differences can be discovered between semantic data models and other data models (e.g. relational and entity-relationship data models). This is not a surprise: earlier hierarchical and network data models already enabled us to find solutions for such situations.

The advantages of generalization appear especially in situations with many irregularities, alternatives or exceptions. These new aspects were not fully covered by other data models. It gave semantic models a clear advantage over other existing data models.

In this paper the advantages of semantic abstractions are further extended. Although semantic concepts enable us to create data models for ordering and sequencing, they are hardly mentioned in literature. This is also caused by the difficulties data models have with modeling time aspects. It is also illustrated with the phrases *is-part-of* and *is-a*, in other data models often used instead of aggregation and generalization. The general need for this extended functionality appeared also in papers discussing characteristics that must be satisfied by the next generation of database systems [4, 10].

Sequencing is also important in the area of object oriented databases. Examples can be found in literature [e.g. 1, 3, 5, 7, 8, 17]. However, often extended entity-relationship diagrams, flow diagrams or event-condition-action diagrams are used for the purpose. These specifications are far from complete; they must be accompanied with many informal procedural descriptions. These specifications/descriptions are inadequate for development of the required software.

Several commercial products support some form of version management (for example: Objectivity/DB, Itasca, ObjectStore, Ontos and Versant). However, a standard set of features for version management is lacking [3, 6].

This paper presents extensions in the usage of semantic abstractions. Both aggregation and generalization can be used for modeling ordering and sequencing. Because time is involved, also the phrases *was-part-of* and *was-a* should be used in stead of only the phrases is-part-of and is-a. The new opportunities will be illustrated with several practical examples.

The resulting high-level specifications can be used for standardization purposes. They can be implemented in any programming environment (e.g. C or C++) or database environment (e.g. relational or object oriented).

For a better understanding of the opportunities, first a short introduction to the underlying semantic concepts [14] is given.

2 Abstractions

This section contains a global overview of the concepts for semantic data modeling using well-known examples. Each object will be visualized explicitly by clearly

distinguishing between identification and descriptive properties. The resulting data models gain in semantic contents as a consequence, while ambiguities and contradictions in the specification are avoided. Only three fundamental abstraction types with clear graphical equivalences in the structural diagrams are required to guarantee inherent semantic integrity. They make use of the fundamental *type-attribute relationship*.

The real world is described by considering types of relevant objects, a type being defined as a fundamental notion. The abstraction leading to a type is called **classification**. The examples (i.e. instances) occurring in a database and required for the recognition of a type are purely illustrations of the concept. The type is *not* being defined hereby. Types are represented by rectangles in diagrams, see figure 1. The counterpart of classification is called **instantiation**.

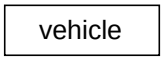

Figure 1: Classification

Aggregation is defined as the collection of a certain number of types in a unit, which in itself can be regarded as a new type (note the analogy with the mathematical set concept). A type occurring in an aggregation is called an attribute of the new type.

Aggregation allows view independence: we can discuss the obtained type (possibly as a property) without referring to the underlying attributes. By applying this principle repeatedly, a hierarchy of types can be set up. An example is given in figure 2. Normally the hierarchy contains only aggregated types.

Aggregation is indicated by a line connecting the centers of two facing rectangle sides, while the aggregate type is (according to its definition) placed above its attributes. Of course, aggregation also has its counterpart: the description of a type as a set of certain attributes is called **decomposition**.

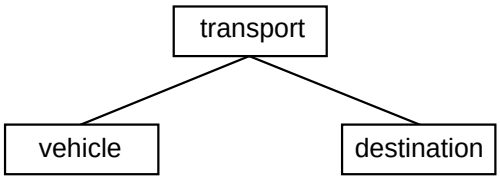

Figure 2: Aggregation hierarchy

A type is defined by listing its attributes, so we could have the following type definitions:

type transport = vehicle, destination,
delivery_date, cargo.
type vehicle = manufacturer, model, price, fuel,
construction_year.
type destination = client_name, address, city,
telephone number.

An example to illustrate the database contents is table 1.

transport	vehicle	destination	delivery_ date	cargo
t1	v1	d2	19961206	paper
t2	v3	d4	19961207	milk

Table 1.

Type definitions carry semantics; they contain the essential properties (e.g. uniqueness of the identifications t1 and t2 in the table above) and essential relationships (the related vehicles v1 and v3 and the related destinations d2 and d4 must occur in related tables). Aggregation can be described using the verb *to have*. According to the above type definition, a vehicle has a manufacturer, model, price, fuel and construction_year. Identifications are properties denoted by type names (see table 1 above). This interpretation implies singular identifications. Attributes (not types!) may contain *roles*. An example is 'construction_year' related to type 'year'. Roles are separated from the type by an underscore. Spaces are irrelevant in type definitions.

The third kind of abstraction, important to conceptual models, is called **generalization**; it is defined here as the recognition of similar attributes from various types and combining these in a new type (note the analogy with the intersection operation from mathematical set theory). We can equally discuss the new type without mentioning the underlying attributes, and it can in itself again serve as a property (i.e. it allows view independence).

Example: consider manufacturer, model, price, fuel, construction_year, cabin, weight, wheels, power and coupling. The corresponding type is truck. Consider, in addition, manufacturer, model, price, fuel, construction_ year, chassis, seats and color, where the type might be car.

The common attributes of the two types are: manufacturer, model, price, fuel, construction_ year. If required, these attributes result in a new type 'vehicle', which may be regarded as the generalization of truck and car. Generalization can be represented in abstraction hierarchies, as we have seen in the case of aggregation. This is shown in figure 3.

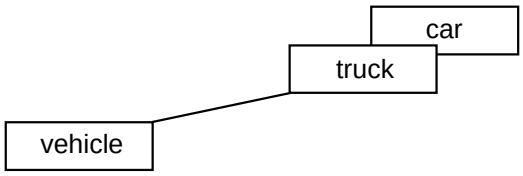

Figure 3: Generalization hierarchy

In abstraction hierarchies, generalization is schematically represented by a line connecting facing corners of rectangles, the generalized type being placed below the

specialized ones. Generalization's counterpart (i.e. the union of attributes from different types) is called **specialization**.

In figure 3 we placed truck and car one behind the other, while only one line connects to vehicle. This is the usual representation of disjoint specializations - i.e. a vehicle can be either a truck or a car, but not both. The combination of a group of disjoint specializations is called a *block*, so truck and car constitute a block. Not all vehicles need to be specialized; an example would be a motorcycle occurring only as instance of vehicle.

The generalization, together with the attributes to be added to it, is (by definition of the concept) described in the type definition of the specialization. So the type definitions are:

type vehicle = manufacturer, model, price, fuel, construction_year.
type truck = [vehicle], cabin, weight, wheels, power, coupling.
type car = [vehicle], chassis, seats, doors.

An example of the database contents is given in table 2. This structure imposes uniqueness of attributes related to generalizations. Besides that, values for these attributes may occur only once in a block (i.e. v1 and v3 may not occur as values in the corresponding truck table).

car	[vehicle]	chassis	seats	doors
c1	v1	self-contained	4	5
c2	v3	separate	2	2

Table 2.

Generalization is commonly associated with the verb *to be*. According to the above type definitions, a truck *is a* vehicle with cabin, weight, wheels, power and coupling, while a car *is a* vehicle with chassis, seats and color. The introduction of new identifications for specializations (e.g. c1 and c2 above) makes generalization hierarchies *nontransitive dependent* [9].

The introductory definitions of aggregation and generalization above have already clearly demonstrated the hierarchical character of these abstractions. This will be elaborated in the following sections with an emphasis on ordering aspects.

3 Ordering

Consider the relationship between employees and their manager in an organization. The aggregation hierarchy in figure 4 would be appropriate in case each employee has exactly one manager. This structure corresponds with the following type definitions:

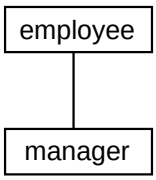

Figure 4: Simple ordering

type manager = name, address, city, salary.
type employee = name, address, city, salary, manager.

The aggregation hierarchy imposes a natural ordering on the data: each employee has one manager, so a manager must exist before employees can be assigned to the manager. This ordering is imposed by the aggregation structure and can also be used in rules or constraints. For example, consider a salary constraint such that employees earn less than their manager. This can simply be formulated as follows:

assert employee *its* salary constraint (*true*) =
salary < manager *its* salary.

We consider now the more general situation in which not just two levels but any number of aggregation levels are concerned. The situation requires a recursive solution as given in figure 5.

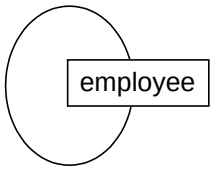

Figure 5: Recursive ordering

The recursive hierarchy requires only one type definition as follows:

type employee = name, address, city, salary, manager_employee.

Because the top manager has no predecessor, the value for manager_employee must here be NULL or another default value. Constraints are now similar. For example the previous salary constraint is formulated as follows:

assert employee *its* salary constraint (*true*) =
manager_employee = NULL *or*
salary < manager_employee *its* salary.

This recursive hierarchical structure can also be used for ordering. Suppose a software project results in an ordering of different versions of software modules. A similar structure can be used in this case. For example, the following type definition could satisfy in case each version has one predecessor:

type version = description, completion_date,
previous_version.

This structure permits successive versions of software: each version has one predecessor and may have several successors. An additional constraint could now be:

assert version *its* time constraint (*true*) =
previous_version = NULL *or*
completion_date >
previous_version *its* completion_date.

4 Sequencing

Specialization can be used in case of a number of successive events. This section contains an example to illustrate the use of abstractions to offer an effective means of progress control.

Suppose data about software projects must be registered. A project can be in one of the following stages:

- *open*: it is defined and can be started;
- *ongoing*: an employee has been assigned;
- *completed*: it is terminated with a report.

In each stage a number of properties must be registered. At the end it must be possible to evaluate the progress of the whole project from start to completion. The first alternative for this situation is to create one single type in which all (present and future) properties are registered. This type definition looks as follows:

type project = subject, employee, start_date,
completion_date, report_title, assessment,
reviewer, stage.

This solution has a number of drawbacks:

- some attributes have no meaning in certain stages, for example: what is the meaning of assessment in an ongoing project?
- some attributes receive different meanings in different project stages, for example: start date is a planned date for an open project while at completion stage it is a realized date.
- some attributes receive a meaning at the same time, for example: report title and assessment are registered at completion time, before completion they have no meaning.

A better solution is found using specialization. According to the different stages it is obvious to distinguish between: (open) project, ongoing project and completed project. Ongoing project and completed project can be considered as disjoint specializations of project. This results in figure 6. The following type definitions correspond with this figure:

type project = subject, planned start_date.
type ongoing project = [project], start_date,
planned completion_date, employee.

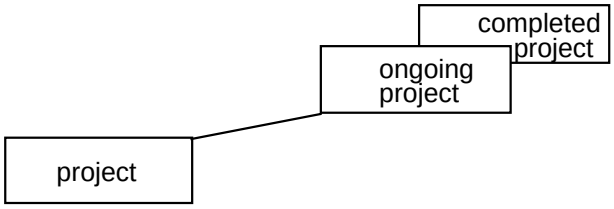

Figure 6: Preliminary sequencing

type completed project = [project], start_date,
completion_date, employee, assessment,
reviewer, report_title.

This solution still offers the possibility of registration of attribute values without meaning. Both ongoing and completed project have the property of being a project, and a project cannot be both ongoing and completed at the same time. However, a project can only be completed when it first has been removed as ongoing project (they are mutually disjoint specializations). This solution has therefore certain disadvantages:

- certain attributes must be transferred from ongoing to completed project (for example: start_date and employee);
- parts of the history are removed by transfer of data (for example: planned completion date).

The recognition of the relationship between open and ongoing project resulted in an improvement. The relationship between ongoing and completed project is analogous to this. A better alternative consists of a structure in which completed project is considered as a specialization of ongoing project. Historical data on attributes of ongoing project do not disappear anymore. This solution is presented in figure 7.

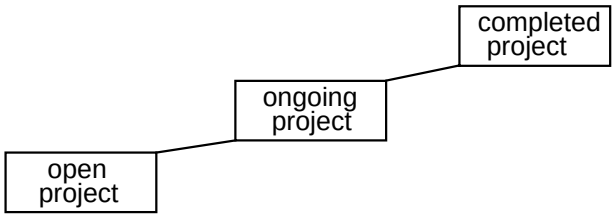

Figure 7: Final sequencing

The corresponding type definitions are now:

type open project = subject, planned start_date,
stage.
type ongoing project = [past_open project],
start_date, planned completion_date,
employee.
type completed project = [past_ongoing project],
completion_date, assessment, reviewer,
report_title.

The solution above has certain advantages:

- the complete history of the project can be determined at any time;
- there are no attributes without meaning, every attribute receives its meaning at the right moment;
- the order of events is completely determined by the given structure;
- transfer of data from one type to another type is no longer necessary, data from earlier stages are accessible from later stages.

This modeling of events is therefore preferred in case of a fixed number of successive events. Specialization is still related to the verb *to be*. However, now we have to use the *was-a* phrase instead of the *is-a* phrase. So a completed project *was an* ongoing project and an ongoing project *was an* open project.

In case an unknown number of project stages is involved, the recursive solution of figure 8 would be appropriate.

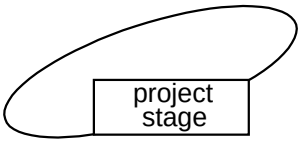

Figure 8: Recursive sequencing

The corresponding type definition could be:

type project stage = [previous_project stage],
subject, employee, report, start_date,
completion_date.

This recursive solution is generally not allowed in object oriented data models [2]. Although inheritance of structure and data is still a relevant matter, the inheritance of methods appears to be nonsense because it is not useful to inherit something that is already encapsulated.

5 Version management

Previous simple cases can be used as a starting-point for more complex models in which both ordering and sequencing occur. Consider for example data about ongoing software projects. Software projects start with a design stage in which functionality and feasibility of the new product are studied. Such a design stage results in a report in which details about the required software modules are described. Each software module has to perform a certain task and together these modules form a configuration of the product. In case of a positive assessment and a management commitment, the next project stage can be initiated. A team is created that will realize the product. In this stage, several prototypes (versions) of the software modules are developed. Versions can be evaluated and result in improvements based on earlier versions. Finally, when a configuration meets all requirements, it can be released as a new product. In that situation a maintenance team keeps responsibility for the product. In between, developers can still work on new versions that may lead to other new releases and so on.

From the foregoing description can be concluded that the following major project stages can be distinguished:

- *design*: a team is created to study functionality and feasibility, this stage can be split up into the substages 'start design' and 'completed design' if desired;
- *realization*: a team is created to construct the desired product, this is considered as a continuous process;
- *maintenance*: a team is created that will be responsible for maintenance of the product, also this stage can be split up into the substages 'start maintenance' and 'completed maintenance' if desired.

These stages can be modeled using the previous project sequencing example (see figure 7).

Several versions of the product appear during realization, it implies usage of previous ordering example, i.e.:

type version = description, realization stage,
state, previous_version.

The type version has the following attributes: description, realization stage (containing a reference to the realization stage of the project), state (indicating the state in which the version occurs: transient, working or released) and previous_version (indicating the relationship with its predecessor). However, in this case an additional complication may occur. The earlier ordering solution allows only several successors of one single version (i.e. branching versions) but not one single successor of several versions (i.e. merged versions). An example is figure 9.

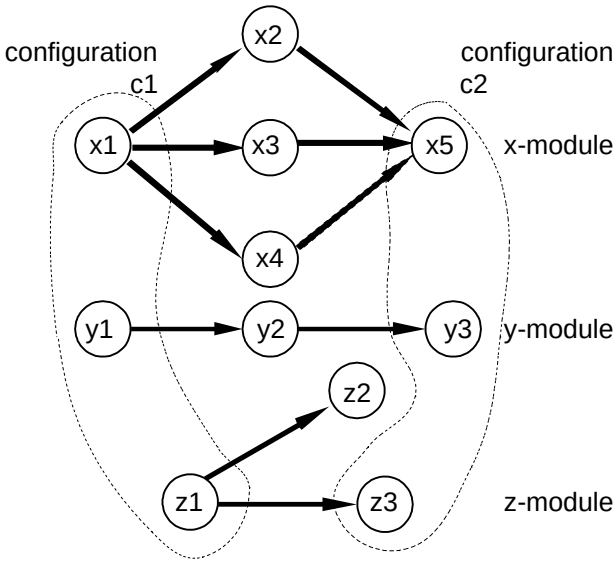

Figure 9: Version histories

The corresponding database contains at least the relationships of table 3 for versions in figure 9.

version	.....	previous_version
x1	.	NULL
x2	.	x1
x3	.	x1
x4	.	x1
x5	.	x3
y1	.	NULL
y2	.	y1
y3	.	y2
z1	.	NULL
z2	.	z1
z3	.	z1

Table 3.

It is clear that situations occurring in the y-module and z-module (i.e. strict sequential ordering and branching) are allowed by the recursive structure. However merging, as occurring in the x-module (x2, x3 and x4 resulting in x5), is not allowed. So the solution requires an extension. The following alternatives can be considered (see figure 10).

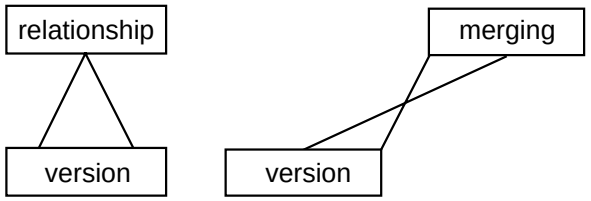

Figure 10: Alternatives for merging

These two alternatives have certain characteristics:

a. *type* relationship = previous_version, next_version.
This solution is most obvious and ignores the required ordering. This type definition allows not only useful but also useless combinations (e.g. combination (x5, x1) in figure 9). This problem can be left to the responsibility of the user. An alternative is to implement many complex constraints to restrict this collection of relationships to only useful ones.

b. *type* merging = [previous_version], merged_version.
This type definition allows each version to have more than one previous version, with the restriction that each version can occur at most once in a merging. Besides that, both previous_version and merged_version must already occur in the database (referential integrity). The following constraint prevents that version relationships (by attribute previous_version) occur in both version and merging:

assert merging *its* uniqueness (*true*) =
previous_version ≠
merged_version *its* previous_version.

This type definition accepts now only those relationships that caused the merge and not those already occurring in the version table. So, the corresponding database for figure 9 contains the data of table 4.

merging	[previous_version]	merged_version
m1	x2	x5
m2	x4	x5

Table 4.

The remainder of this problem has to do with the modeling of configurations. Again, two alternatives can be considered (see figure 11).

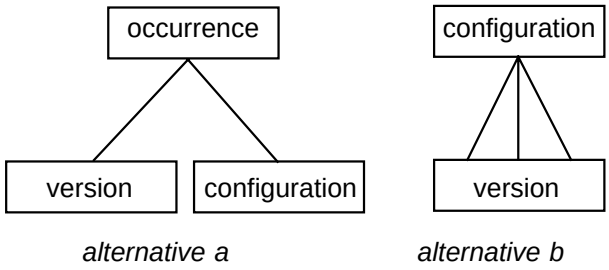

Figure 11: Configuration alternatives

The relevant type definitions are now as follows:

a. *type* occurrence = version, configuration
This structure represents an *n* to *m* relationship between configuration and version, so a version may occur in several configurations and a configuration consists of several versions. This is a weak reflection of the restrictions in reality because:
- not each number of versions is allowed in a configuration (e.g. in figure 9 exactly three different versions);
- not each combination of three versions is allowed in a configuration (e.g. the combination x1, x2 and y3 in figure 9 is not allowed).

Many complex constraints must be imposed on this structure to prevent these situations.

b. *type* configuration = x_version, y_version, z_version.
This structure fits perfectly for the example in figure 9. Additional constraints are also very easily imposed on this simple structure. For example, to be sure that all versions are in the working state (i.e. become

visible to other users) simple constraints like the following are required:

assert configuration *its* x_working (*true*) =
x_version *its* state = "working".

A representation for the previous configurations (including the ordering relationship between configurations) results in table 5.

config-uration	x_ version	y_ version	z_ version	previous_ configuration
c1	x1	y1	z1	NULL
c2	x5	y3	z3	c1

Table 5.

Finally, a release is a special configuration in the sense that it is supported by a maintenance team. Release can therefore be considered as a specialization of configuration. So we have also:

- *configuration*: prototype of the product with predecessor. All versions become visible to other users;
- *release*: final prototype of the product turned over to a maintenance team. All versions in a configuration are frozen for use.

This leads to the global structure of figure 12. With this structure correspond the following type definitions:

type design stage = project_name, start_date,
study_team, report, assessment.
type realization stage = [past_design stage],
development_team, start_date.
type maintenance stage = [past_realization stage],
maintenance_team, start_date.
type version = description, realization stage,
state, previous_version.
type merging = [previous_version],
merged_version.
type configuration = x_version, y_version,
z_version, previous_configuration.
type release = [configuration], maintenance stage,
release_date.

Discussion

Although the global structure in figure 12 may look self-evident, it needs further explanation. Consider the modeling of version. Each version has a relationship with the realization stage. Versions also have an ordering relationship (from the attribute previous_version). Because of these two relationships, the structure above allows different values for realization stage in case of successive versions. This is not acceptable and to prevent this, the following additional constraint is needed:

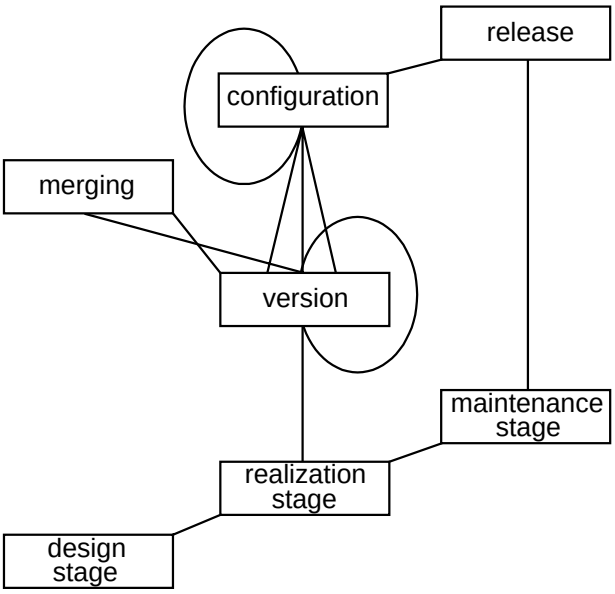

Figure 12: Project management

assert version *its* correct realization (*true*) =
previous_version = NULL *or*
realization stage =
previous_version *its* realization stage.

Another situation occurs frequently in data models. It is caused by multiple inheritance paths between different types. Consider the type release in figure 12. Release has two different paths leading to realization stage. The path from realization stage to maintenance stage is needed to assure that the release will be turned over to the maintenance team. It is evident that in both cases the same realization stage is referred to (note that this is not generally the case). This implies that constraints like the following must be added to the structure:

assert release *its* x_allowed (*true*) =
configuration *its* x_version *its* realization stage =
maintenance stage *its* realization stage.

Simple constraints like the following are needed to require that all versions are frozen for use by others:

assert release *its* x_frozen (*true*) =
configuration *its* x_version *its* state =
"released".

An advantage of the semantic hierarchy has been illustrated with this complex example. Because the position of a type in the abstraction hierarchy is completely determined by the role in aggregation/generalization definitions, derivable relationships are immediately visible. It is therefore very simple to add necessary constraints using these derivable relationships. A data model

without constrained hierarchies would make it much more difficult to find additional constraints.

Implementation

The semantic structure determines the basic functionality for sequencing and ordering in an application environment. All concepts in the semantic structure (for example in figure 12) can be implemented using simple file structures (c.f. foregoing tables). It is evident that it must be possible to create, destroy and modify instances in these files. The explicit presence of all concepts in the semantic structure facilitates implementation. Inherent constraints (uniqueness and valid references) must be added to each operation. All other constraints must result in triggered procedures that must be incorporated into these operations. In a relational database environment this can be realized by using the concepts of primary, foreign and candidate key; in an object oriented database environment constraints can be incorporated in object methods. Methods can also be used for further tailoring to the needs of the user's environment. Query languages, screen generators and browsers can provide traversal of the underlying relationships.

6 Conclusion

It has been demonstrated that the semantic abstractions classification, aggregation and generalization are suitable for modeling event ordering and sequencing. Necessary constraints result in small pieces of additional software. This has certain advantages over a procedural approach.

Exact, high-level and unambiguous specification of ordering by means of data model relationships leads to meta information which enables a database management system to derive the required checks. This specification of ordering concepts can also be used for standardization purposes. These properties make semantic data models a candidate for usage in the next generation of database management systems. The experimental Xplain system [15, 16] has already demonstrated many of these advantages.

Acknowledgments

I wish to thank my colleagues Bert Bakker and Dolf van der Ende for frequent discussions on semantic data modeling and meta modeling. I thank also all graduate students who participated in the development of the Xplain DBMS.

References

[1] D. Beech and B. Mahbod, Generalized version control in an object-oriented database, Proceedings IEEE Fourth Int. Conf. on Data Engineering, (1988), pp. 14-22.

[2] E. Bertino and L. Martino, Object-oriented database management systems: concepts and issues, IEEE Computer 24 (1991) 4, pp. 33-47.

[3] R.G.G. Catell, Object data management (Revised Edition), Addison-Wesley, Reading MA (1994).

[4] The Committee for Advanced DBMS Function, Third-generation database system manifesto, SIGMOD Record 19 (1990) 3, pp. 31-44.

[5] T. Connolly, C. Begg and A. Strachan, Database Systems: a practical approach to design, implementation and management, Addison-Wesley, Reading MA (1995).

[6] R.H. Katz, Toward a unified framework for version modeling in engineering databases, ACM Computing Surveys 22 (1990) 4, pp. 375-408.

[7] W. Kim and H.T. Chou, Versions of schema for object-oriented database systems, Proceedings Int. Conf. on Very Large Databases VLDB 1988, (Sept. 1988).

[8] W. Kim and F.H. Lochovsky (eds.), Object oriented concepts, databases and applications, Addison-Wesley, Reading MA (1989).

[9] J.P. Rosen, What orientation should Ada objects take?, Communications of the ACM 35 (1992) 11, pp. 71-76.

[10] A. Silberschatz, M. Stonebraker, J. Ullman (eds.), Database research: achievements and opportunities into the 21st century, Report of an NSF Workshop on the Future of Database Systems Research, May 26-27 1995, http://db.stanford.edu/pub/ullman/lagii.ps.

[11] J.M. Smith and D.C.P. Smith, Database abstractions: aggregation, Communications of the ACM 20 (1977) 6, pp. 405-413.

[12] J.M. Smith and D.C.P. Smith, Database abstractions: aggregation and generalization, ACM Transactions on Database Systems 2 (1977) 2, pp. 105-133.

[13] J.M. Smith, Principles of database conceptual design, Proceedings NYU Symp. on database design (1978), pp. 35-49.

[14] J.H. ter Bekke, Semantic data modeling, Prentice Hall, Hemel Hempstead (1992).

[15] J.H. ter Bekke, Complex values in databases, Proceedings Int. Conf. on Data and Knowledge Systems for Manufacturing and Engineering DKSME 1994, Hong Kong (1994), pp. 449-455.

[16] J.H. ter Bekke, Meta modeling for end user computing, Proceedings Workshop on Data and Expert Systems Applications DEXA 95, London (1995), pp. 267-273.

[17] W. Tichy, Revision control system, Proceedings IEEE Sixth Int. Conf. on Software Engineering, (Sept. 1982).