SEED - A DBMS FOR SOFTWARE ENGINEERING APPLICATIONS BASED ON THE ENTITY-RELATIONSHIP APPROACH

Martin Glinz Jochen Ludewig *)

Brown Boveri Research Center
CH-5405 Baden, Switzerland

ABSTRACT

SEED is a database system which supports the data engineering needs of a software engineering environment. It provides information structures that are not incorporated in conventional database systems, but are typical in the software engineering process.
This paper describes two principal features of SEED: how to deal with vague and incomplete information without giving up consistency checking, and the management of database versions and variants.
A prototype of SEED is used as the database for an existing specification and design tool.

INTRODUCTION

A software engineering environment uses information structures that are rather different from those provided by conventional database systems. Building a DBMS for software engineering applications therefore requires the development of new, engineering oriented database concepts.

Existing work on semantic database modelling and on engineering databases (for an overview, see [12], [6] and the references cited there) provides solutions to many points in the problem space. However, we found no work addressing the full scope of database requirements when developing tools for software specification and design. On the other hand, software engineering is a field large enough to justify tailored solutions.

On this background, we designed SEED (which stands for Software Engineering Environment Database System). A prototype of SEED was implemented in a straightforward manner, deriving the implementation concepts from the model.

Our ultimate goal was not to invent a new database model, but to provide a DBMS that substantially eases the task of data engineering when building a software engineering environment consisting of a set of cooperating tools.

*) with ETH Zurich since January, 1986

CONCEPTS

Concepts for software design

The concept of SEED was strongly influenced by our work on the specification system SPADES [9] and its predecessor, ESPRESO. We therefore briefly outline our approach to software design.

We consider specification and design to be evolutionary, strongly intertwined processes. Their goal is to model the target software system. Such a model is a semiformal description of the objects and relationships that the target system is composed of.

Development starts with informal, incomplete, and vague textual descriptions and evolves to a rather formal representation by objects and relationships of well defined sorts. Information is accepted independently of its formality and completeness. But at any stage, the collected information must be consistent (according to the semantics of a specification grammar). Eventually, the result must be sufficiently formal, complete, and precise to serve as a basis for implementation.

The state of the development is saved after every larger modification. Rollback to prior states or tracing alternatives allow for exploring the design space and for undoing errors.

Basic ideas of SEED

SEED is based on the entity-relationship approach [2]. This approach is especially suited for software development with semiformal models. However, the entity-relationship model lacks some features that are vitally needed in a DBMS for software engineering applications: object hierarchies, a sophisticated consistency concept, how to deal with vague or incomplete information, and management of versions and variants.

In SEED, these features are added to the entity-relationship model [7]. Figures 1 and 2 give a general idea of the basic concepts: Figure 1 shows some objects and relationships that are handled by SEED using the schema of figure 2. This schema describes the data model of a primitive specification system where actions, data, and data flow may be

represented. We use modified entity-relationship diagrams for graphic representation.

Notice the difference between figures 1 and 2: Figure 1 gives an example of data that may be stored in SEED. Figure 2 shows a SEED-schema that defines what kinds of data may be stored.

In this paper, we focus on two extensions that we consider most important: (1) the problem of admitting vague and incomplete data without loosing consistency control and (2) management of versions and variants.

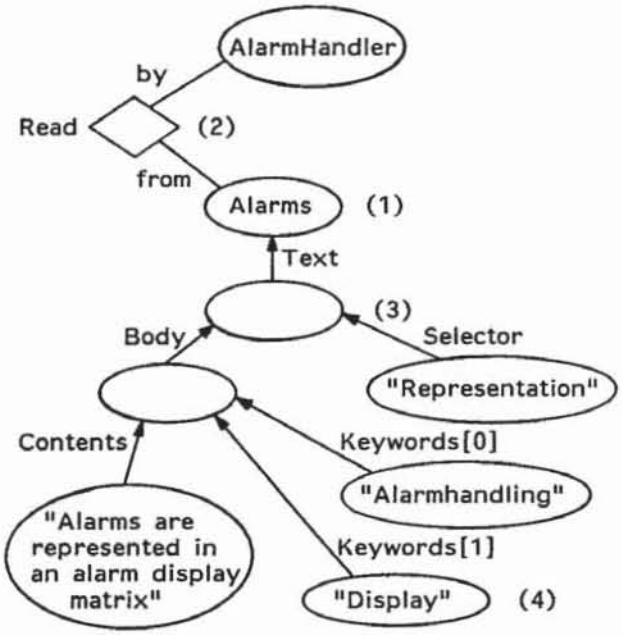

Figure 1: Sample object-relationship structure

Explanation of figure 1:
(1) is an independent object with name 'Alarms'. (2) is a relationship 'Read', relating objects 'AlarmHandler' and 'Alarms' in roles 'by' and 'from', respectively. All objects below of 'Alarms' are dependent objects (sub-objects). The name of a dependent object is composed of the name of its parent and of its role in the context of the parent object. Thus, (3) is the object 'Alarms.Text' consisting of objects 'Alarms.Text.Body' and 'Alarms.Text.Selector'. The latter has the value "Representation". Finally, (4) is a dependent object with name 'Alarms.Text.Body.Keywords[1]' and with value "Display".

Explanation of figure 2:
'Data' is a hierarchically structured object class with class 'Data.Text' as a subclass, which again has the subclasses 'Data.Text.Body' and 'Data.Text.Selector'. The latter has objects of type STRING as instances. 'Data.Text' has the cardinality 0..16, specifying that any object of class 'Data' may have from zero up to 16 objects of class 'Data.Text'. Classes 'Data' and 'Action' are related by associations (relationship classes) 'Read' and 'Write' with cardinalities 1..* and 0..*. '1..*' means that any instance of 'Data' must have at least one 'Read'- ('Write'-) relationship with an instance of 'Action'; there is no upper bound for the number of such relationships. The roles 'from' and 'by' of the 'Read'-association express that reading is from instances of 'Data' by instances of 'Action'. The association 'Contained' imposes a tree structure on the objects that are instances of 'Action' by means of the attribute ACYCLIC and the cardinality 0..1 for the role 'in'.

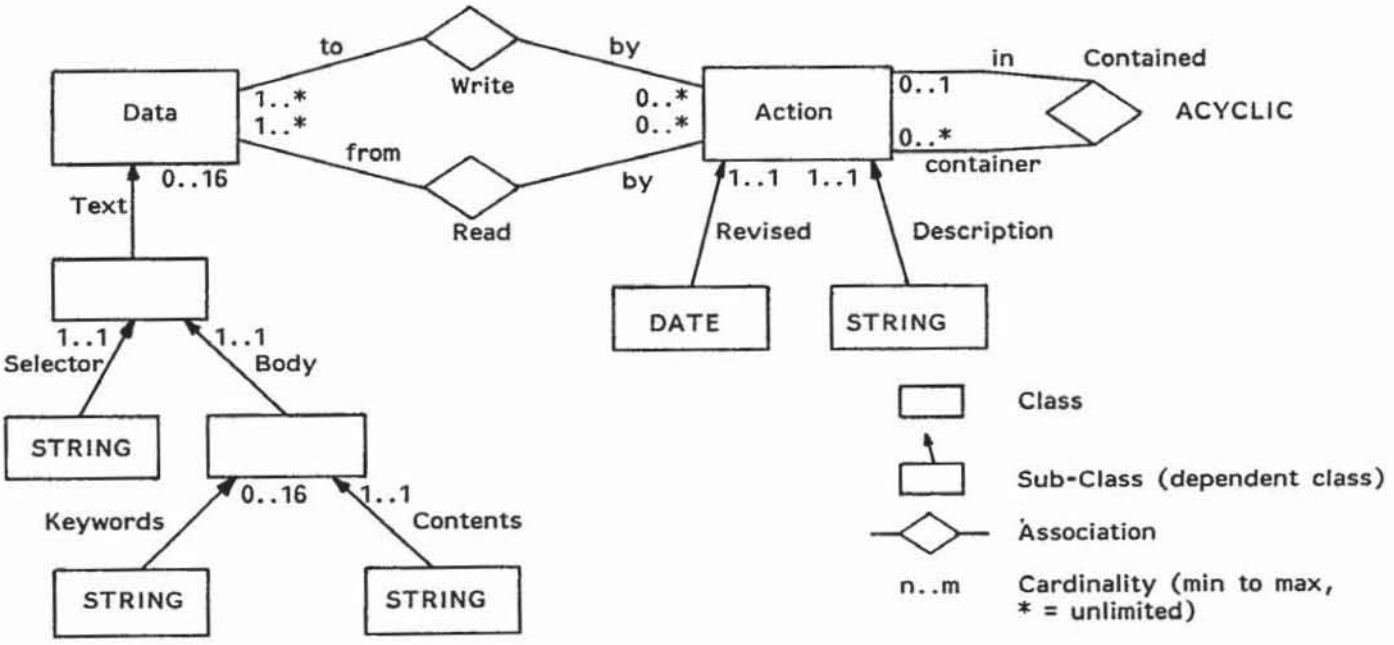

Figure 2: A sample SEED schema

MANAGING VAGUE AND INCOMPLETE INFORMATION

The normal approach to database consistency is to require all data in the database to fully comply with the structures and constraints given in the schema. However, this approach prevents the entry of incomplete and vague information into the database.

We use the schema of fig. 2 for two examples:

(1) We cannot store the information that there is a dataflow from 'AlarmHandler' to 'Alarms' unless we precisely know whether it is a read or a write, because there is no schema category which fits the vague information about the existence of a dataflow.

(2) We cannot enter 'Alarms' as an object of class 'Data' without also entering a 'Read'- and a 'Write'-relationship of 'Alarms' with objects of class 'Action', because the database would become inconsistent otherwise. This is due to the fact that the minimum cardinalities of the 'Read' and 'Write' associations require every object of class 'Data' to have at least one 'Read'- and one 'Write'-relationship with objects of class 'Action'.

Management of vague and incomplete data therefore requires extended schema structures as well as a modified consistency concept.

Vague data

Generalization is a well known principle for representing meta-classifications ('is-a'-relationships) [11]. This principle can be used to define categories in the schema that allow for dealing with vague data in a well defined manner. We extend generalization from object classes also to associations (relationship classes).

Wherever we want to allow for vague information, we define a hierarchy of generalizations: Generalized classes and associations provide categories to enter vague data. When the knowledge about these data becomes more precise, they are moved down in the generalization hierarchy to one of the specializations.

Figure 3 shows an example: The schema of fig. 2 is modified such that associations 'Write' and 'Read' are generalized to 'Access'. Class 'Data' is specialized (inverse of generalization) to 'OutputData' and 'InputData'. Classes 'Data' and 'Action' are generalized to 'Thing'.

This allows storage of vague information like "There is a thing with name 'Alarms'". When we know more about 'Alarms', e.g. that it is a data object which is accessed by action 'Sensor', we may make the previously stored information more precise by re-classifying 'Alarms' in class 'Data' and introducing an 'Access'-relationship with 'Sensor'. In a next step, we might learn that 'Alarms' is an output. Again, we can make the stored information more precise by specializing the 'Access'-relationship to a 'Write'-relationship. Finally, we could arrive at a precise information like "'Alarms' is an output written twice by 'Sensor', and writing is repeated in case of error".

In generalization hierarchies of associations, different cardinalities may be used to express additional semantics. For example, the cardinality 1..* of 'Access by' means that every object of class 'Action' eventually must access at least one object of class 'Data'. However, the cardinality 0..* of 'Read by' and 'Write by' allows either a write or a read access to satisfy this condition.

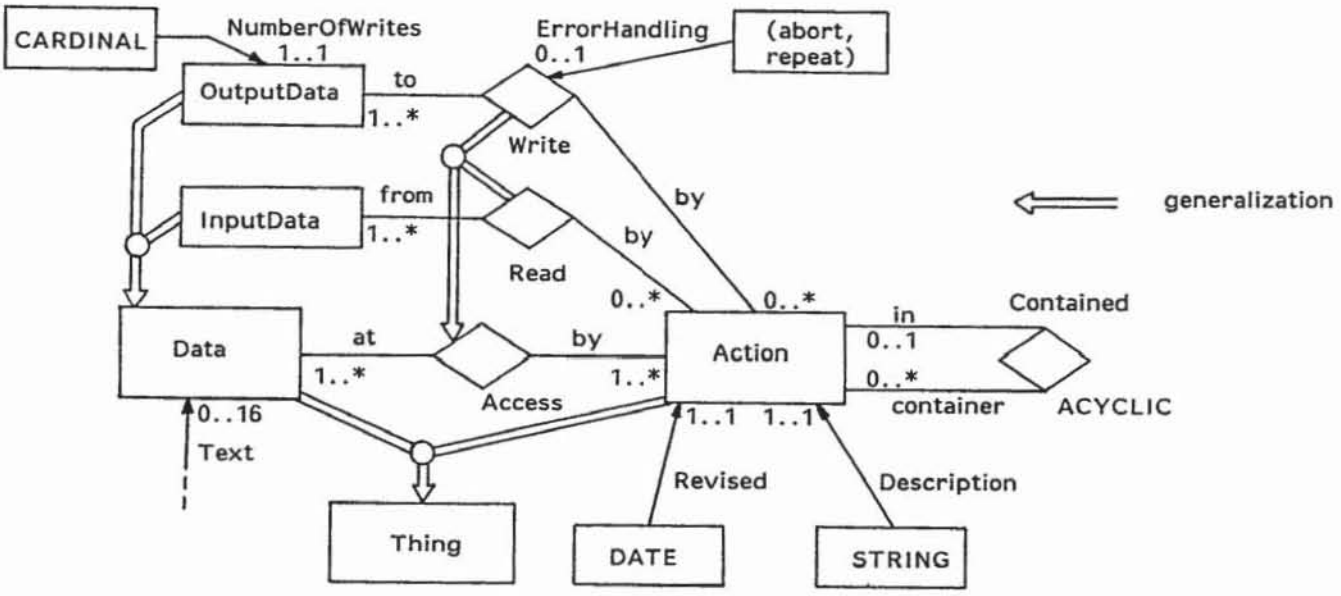

Figure 3: SEED schema with generalizations of classes and associations

Incomplete data

We already mentioned that minimum cardinalities restrict the treatment of incomplete information. However, we do not want to omit minimum cardinalities as they provide information about the desired final state of data that is being stored in the database.

The problem is solved by partitioning the information that is provided by the schema into two categories: consistency and completeness information. Class and association membership, maximum cardinalities, ACYCLIC-conditions, and attached procedures are consistency information. Minimum cardinalities and covering conditions for generalizations represent completeness information.

(Attached procedures may be attached to any SEED schema element. They are executed when an item of the corresponding schema element is updated. Attached procedures are used to express complex integrity constraints. A generalization is covering if every data item must finally be specialized in a specialized class (or association) of this generalization.)

Manipulating vague and incomplete data

Manipulation of vague data requires an operation for re-classifying an existing data item within a generalization hierarchy.

As we allow for incomplete data, we may have objects with undefined sub-objects and not yet existing relationships. The semantics of such objects in database operations is simple: When the database is searched for data that meet certain selection criteria, an undefined object matches nothing. Taking joins or cartesian products is not affected by undefined items. This is due to the fact that entity-relationship based models define these operations on existing relationships only.

Whenever an update operation is executed, SEED checks all consistency rules, that are derivable from the consistency information mentioned above, and that apply to the data being updated. Thus SEED permanently ensures database consistency.

Formal detection of incompleteness is provided by operations which check the rules that are derivable from the completeness conditions in the schema.

VERSIONS AND VARIANTS

Versions

The SEED version concept allows certain states of the database to be preserved. It aims at long term preservation, e.g. when a document has been finished or a product is released, as well as at short term logging, e.g. saving the database state before and after a session. However, SEED does not keep a log of every database update.

Versions are created explicitly by taking a snapshot of the database. Additionally, there is always a current version representing the current state of the database. Every update changes this state, replacing the current version with a new one. When a current version is to be saved, an explicit version generation must be performed prior to the update.

Versions are identified by a decimal classification. The classification tree reflects the version history.

Versions cannot be modified, except for deletion. However, alternatives may be created by selecting a historical version to become the current version prior to the execution of a sequence of update operations. Work then continues on the basis of this version until it is saved with a version creation command and the original current version is selected again.

Retrieval of data from an old version is performed in the same way as retrieval from the current version. The version of interest is selected prior to the execution of retrieval operations (with the current version as a default). SEED defines additional operations for history retrieval and navigation, e.g. 'find all versions of object 'AlarmHandler', beginning with version 2.0'.

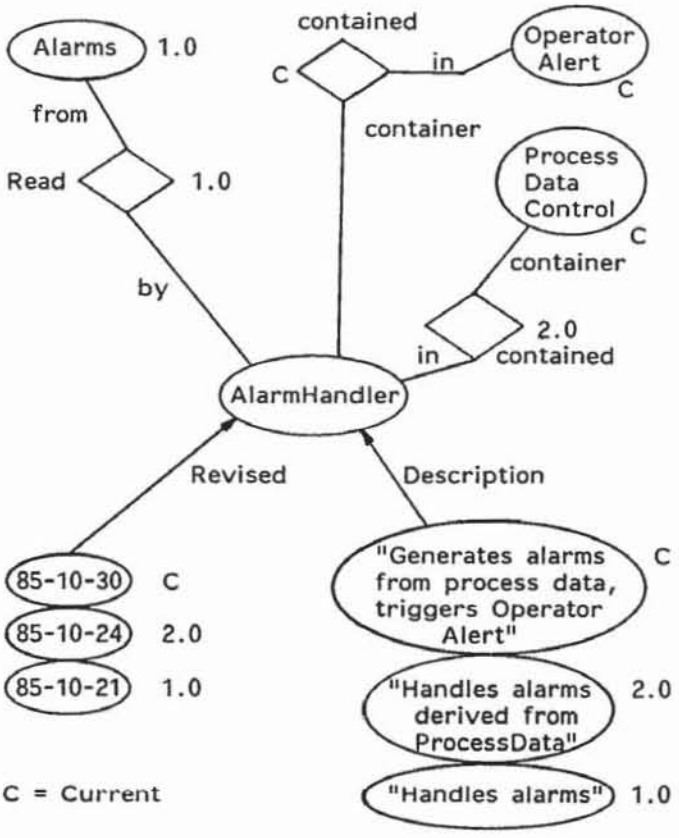

Figure 4a: Sample objects and relationships with versions

When creating a version we do not save the complete database. We only store those objects and relationships that have been changed after the creation of the previous version. Items that have been deleted in this interval must also be recorded. This is made easy by marking items as deleted instead of removing them physically.

Fig. 4 shows an example of objects with multiple versions. The stored versions of an object are represented as a cluster of ovals. The version of a hierarchically structured object is composed of the versions of its sub-objects.

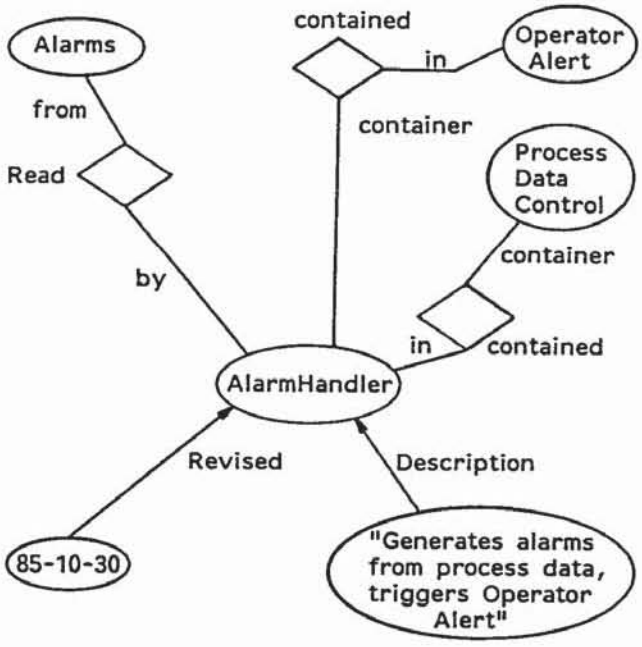

Figure 4b: Current version of data items of figure 4a

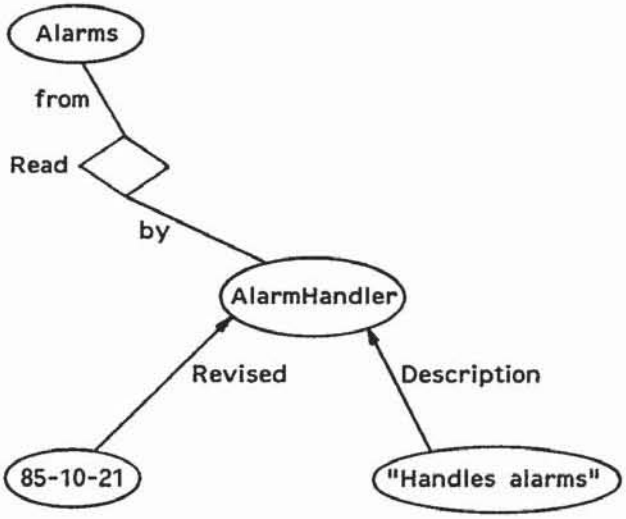

Figure 4c: Version 1.0 of data items of figure 4a

In this example, we have information about three versions: 1.0, 2.0, and Current. From this, we can build views to particular versions. The view to a version with number n consists of the objects and relationships having the greatest version number that is less than or equal to n (provided that they are not marked as deleted). Figures 4b and 4c show the corresponding current version and version 1.0, respectively.

When the schema is modified, the interpretation of versions that were created before this modification becomes a problem. Therefore, we must generate schema versions, too.

Patterns and Variants

When entering information into a database, a user often wishes to express common properties of data that are not reflected explicitly in the schema. For example, the schema may define a class of procedures that are to be specified. A subclass of this class may contain the deadline for the completion of every procedure specification. If a user wishes to express that some procedures have a common deadline and wants to maintain that deadline value consistently for these objects, he/she cannot do so.

In SEED, a pattern concept is provided for dealing with those situations: Any data item that is entered into the database can be marked as a pattern. Patterns are invisible to any retrieval operation and are not checked for consistency unless they are inherited by a 'normal' data item. The semantics of patterns and inheritance is as follows: all retrieval operations view patterns as if they were inserted in the context of the inheritors. However, instead of a real insertion we establish a special inherits-relationship between a pattern and any of its inheritors. Thus pattern information cannot be updated in the context of the inheritors, but only in the pattern itself. Conversely, any update of a pattern automatically propagates to all inheritors of that pattern.

Returning to the example introduced above, the user may define a pattern procedure object with a given deadline. Every real procedure object that should share this deadline, inherits the pattern. The deadline value will be maintained consistently, as it is not changeable in the real objects, whereas a change in the pattern affects all inheriting objects in the same way.

There are several other applications for patterns, e.g. for templates, user defined constraints, or standardized data environments.

Patterns also serve as a basis for managing variants: We define a family of variants to be some sets of objects and relationships that have a part of their information in common, but differ in some other parts. This means that every variant shares a part of its objects and relationships with the other members of the family (the so called common part), but has also objects and relationships that differ from the other members (the variant part).

Variants are different from alternatives: alternatives are coexisting versions of the database, whereas variants express that some information in the database consists of a common part and some varying parts.

An example of variants is a set of system configurations that share most of the software modules, but differ in some hardware dependent modules.

Common and variant parts of a variants family are described by normal items. The connections between the common part and the several variant parts are established by pattern relationships with every variant inheriting these patterns. Pattern semantics now guarantee that all variant parts have the same relationships to the common part. This could not be assured with ordinary relationships.

In fig. 5, the common part is connected to pattern objects PO1 and PO2 by pattern relationships PR1 and PR2, respectively. Both variants inherit these patterns. Thus, they both have inherited relationships to the common part, i.e. they have it in common.

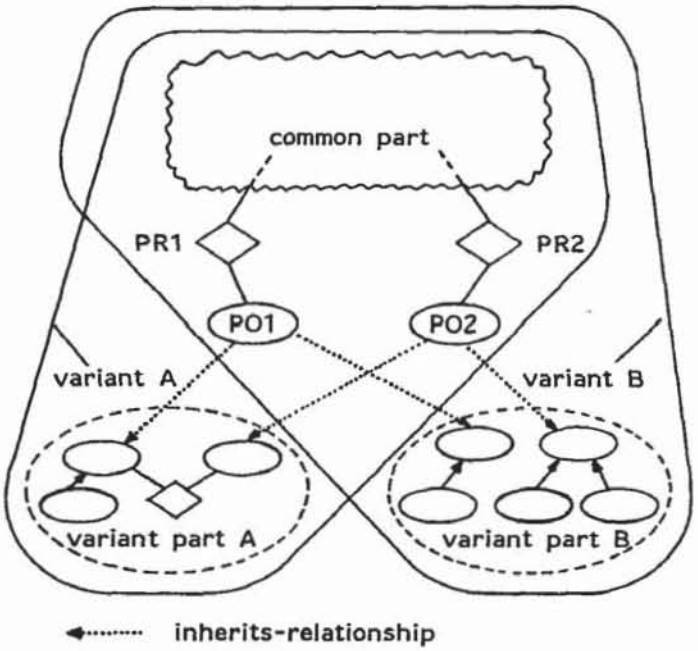

Figure 5: Defining variants by means of patterns

DATA MANIPULATION IN SEED

SEED has been designed to support the data management tasks of software development tools. Hence, SEED has an operational interface that consists of a set of procedures. The SEED prototype provides the procedures for data creation, update, and simple retrieval by name. Retrieval with complex queries is not supported.

RELATED WORK

We should like to acknowledge that SEED incorporates many ideas from work on engineering databases, semantic data models, and extensions of the entity-relationship model.

Smith and Smith [12] deal with a database approach to specification, focussing on formal specifications. Bever and Lockemann [1] also propose an entity-relationship database for a software engineering environment. They concentrate on the coding and compilation phase, where information is fully formalized. Katz and Lehman [8] and Tichy [13] deal with version and configuration management on the level of files.

A semiformal approach to software development with emphasis on the specification and design phase, which SEED aims at, is not covered by this work. The version concept of SEED works on the database, not on files.

The numerous extensions of the entity-relationship model ([3], [4], [5], [10]) point out many solutions to particular problems and have been a valuable source for the design of SEED. However, they reveal no concise solution to the problems of software engineering data management, which is the main goal of SEED.

DISCUSSION

Open problems

SEED is currently a single user system only. The problem of concurrency control and version management in a multi-user environment have not yet been solved. We only have some rough ideas concerning a two level approach: One central server runs the complete database and several clients use the server for retrieval operations, but take local copies for making updates. Data that has been copied to a client for update has a write lock in the central database. When a client sends an updated copy back to the server, the server puts the modified data into the central database in a single transaction. Versions are kept both locally and globally under control of the user and the server, respectively.

In our version concept, we have not yet considered history sensitive consistency rules, i.e. rules that impose constraints for the transition from a given version to its successor.

State of work

A prototype of SEED is operational. It is currently being integrated into the specification system SPADES. Implementation concepts for versions and variants have been developed, but the implementation is not yet done.

The practical use of SEED will give us insight in its benefits and weaknesses. The experience gained

from there will guide the further development of the concepts and the implementation of SEED.

The first experiences with SPADES using SEED show that SPADES has become considerably slower, but much more flexible in the sense that modifications of the system and integration of new features have become much easier.

ACKNOWLEDGEMENTS

We wish to thank Hansjörg Huser and Hans Matheis. Hansjörg implemented the SEED prototype and helped to integrate SEED into SPADES. Hans did the greatest part of the SEED-SPADES integration. We also thank Mark Garrett for commenting on our English.

REFERENCES

[1] Bever, M. and P.C. Lockemann (1984). Database support for software development. In: Morgenbrod, H. and W. Sammer (ed.), Programmierumgebungen und Compiler, Symposium I/84 of the German Chapter of the ACM, Munich. 46-72.

[2] Chen, P.P. (1976). The entity-relationship model - toward a unified view of data. ACM Transact. Database Syst., 1. 9-36.

[3] Chen, P.P. (Ed.) (1980). Entity-relationship approach to systems analysis and design. North Holland. (Proceedings 1st Intern. Conf. on Entity-Relationship Approach.)

[4] Chen, P.P. (Ed.) (1981). Entity-relationship approach to information modeling and analysis. ER Institute, Saugus, Ca. (Proceedings 2nd Intern. Conf. on Entity-Relationship Approach.)

[5] Davis, C.G., S. Jajodia, P.A. Ng, R.T. Yeh (Ed.) (1983). Entity-relationship approach to software engineering. North Holland. (Proceedings 3rd Intern. Conf. on Entity-Relationship Approach.)

[6] Dittrich, K.R., A.M. Kotz, J.A. Müller, P.C. Lockemann (1984). Datenbankkonzepte für Ingenieuranwendungen: eine Übersicht über den Stand der Entwicklung. 14. Jahrestagung der GI, Braunschweig, W.-Germany.

[7] Glinz, M., H. Huser, and J. Ludewig (1985). SEED - a database system for software engineering environments. Proc. GI Conf. on Database Systems for Office Automation, Engineering, and Scientific Applications, Karlsruhe, W.-Germany. (Informatik-Fachberichte vol. 94, Springer Verlag) 121-126.

[8] Katz, R. H., T. J. Lehman (1984). Database support for versions and alternatives of large design files. IEEE Transactions on Software Engineering Vol. SE-10, No. 2. 191-200.

[9] Ludewig, J., M. Glinz, H. Huser, G. Matheis, H. Matheis, M.F. Schmidt (1985). SPADES - A specification and design system and its graphical interface. Proc. 8th Intern. Conf. on Software Engineering, London. 83-89.

[10] Parent, C. and S. Spaccapietra (1984). An entity-relationship algebra. Proc. 1st Intern. Conf. on Data Engineering. 500-507.

[11] Smith, J.M., and D.C.P. Smith (1977). Database abstractions: aggregation and generalization. ACM Transact. Database Syst., 2. 105-133.

[12] Smith, J. M., and D. C. P. Smith (1980). A data base approach to software specification. In W. E. Fairley and R. E. Riddle (Ed.), Software Development Tools, Springer: Berlin-Heidelberg-New York. 176-200.

[13] Tichy, W. F. (1982). A data model for programming support environments and its applications. In: Schneider, H.-J. and A. Wasserman (ed.), Automated Tools for Information Systems Design. North Holland. 31-48.