SEED - A DATABASE SYSTEM FOR SOFTWARE ENGINEERING ENVIRONMENTS

Martin Glinz Hansjörg Huser Jochen Ludewig

Brown Boveri Research Center
CH-5405 Baden, Switzerland

ABSTRACT

A data model for software engineering databases is presented. On the basis of an entity-relationship-approach, the model provides special features for complex objects, consistency, vague or incomplete information, variants, and versions. A prototype of a database system based on this model is being implemented.

ZUSAMMENFASSSUNG

Das Modell einer Datenbank für Anwendungen in der Softwaretechnik wird vorgestellt. Es basiert auf einem Entity-Relationship-Ansatz, enthält aber zusätzliche Modellierungsmittel für komplexe Objekte, Konsistenzsicherung, vage bzw. unvollständige Informationen, Versionen- und Variantenverwaltung. Ein Prototyp eines Datenbanksystems, welches mit diesem Modell arbeitet, wird implementiert.

INTRODUCTION

Every Software Engineering Environment needs a database as a central repository of information (Osterweil, 1981). The requirements for such a software engineering database differ significantly from the capabilities of commercially available database management systems, including relational systems (Stenning, 1983 and Dittrich et al. 1984). The most important requirements are:

(1) storing complex objects (hierarchically structured, undetermined size) and relationships between these objects together with their semantics
(2) storing incomplete or vague information
(3) ensuring database consistency and supporting long transactions
(4) accessing the database in a manner similar to abstract data types
(5) managing versions and variants.

Thus we must either bridge the gap between these requirements and the capabilities of an available database system, or we must implement a new system that meets our requirements. In any case, we must express our needs by defining a conceptual schema and database operations on a high semantic level. However, the available database models offer poor or even no concepts to model a schema and operations in a way that requirements (2) - (5) are met. Therefore we define a new database model that is tailored to software engineering applications. (For details see Glinz and Ludewig, 1984.) In an implementation, this model may either be mapped onto an existing database system or be implemented directly. For our proto-

type implementation, we have chosen a direct, straightforward implementation by means of linked lists. The base abstract machine (see below) of the prototype is operational since November, 1984.

THE SEED DATA MODEL

Our SEED (Software Engineering Environment Database) - Model is based on an entity-relationship approach (Chen, 1976). Instead of entities and attributes, we use the more general concept of hierarchically structured objects. The latter allows for representation of entities with attributes, hierarchical decomposition, and hierarchical ordering.

We include semantics by: (1) a naming concept for objects that resembles Pascal record structures, (2) specifying the roles of the objects in relationships, and (3) defining cardinalities for all schema structures that classify relationships and dependent objects.

Any element in the database schema may have attached procedures that are executed when a data item belonging to that schema element is processed. This allows for an easy definition of dynamic consistency rules and short transactions. So we meet requirement (1).

In object hierarchies, only the top object must be defined. Thus we can store incomplete information. We use generalization (Smith and Smith, 1977) of object classes and of associations (relationship classes) to create categories in the schema that allow to store vague information. So we are able to define and refine information on objects and relationships incrementally (requirement 2).

From information in the schema we derive consistency rules that always must hold, and completeness rules that are checked on demand only. So we can ensure consistency by automatic checks without excluding incomplete and vague data from the database. Long transactions (design steps lasting from minutes to hours) are protected by a write-lockout mechanism (requirement 3).

Access to the database is provided through a set of abstract machines only. The procedures of the lowest level check all consistency rules derivable from the database schema, thus ensuring consistency. As any procedure in a higher abstract machine makes use of procedures in lower ones, consistency is guaranteed for any database access. So the database may be viewed as a set of abstract data types (requirements 4 and also 3).

Our pattern concept (see next chapter) yields an elegant solution to the problem of managing variants. Furthermore, we support a history of versions for objects and relationships by providing database operations that attach version tags to data, and reconstruct or delete versions on demand (requirement 5).

Figure 1 shows a graphic representation of a SEED-schema.

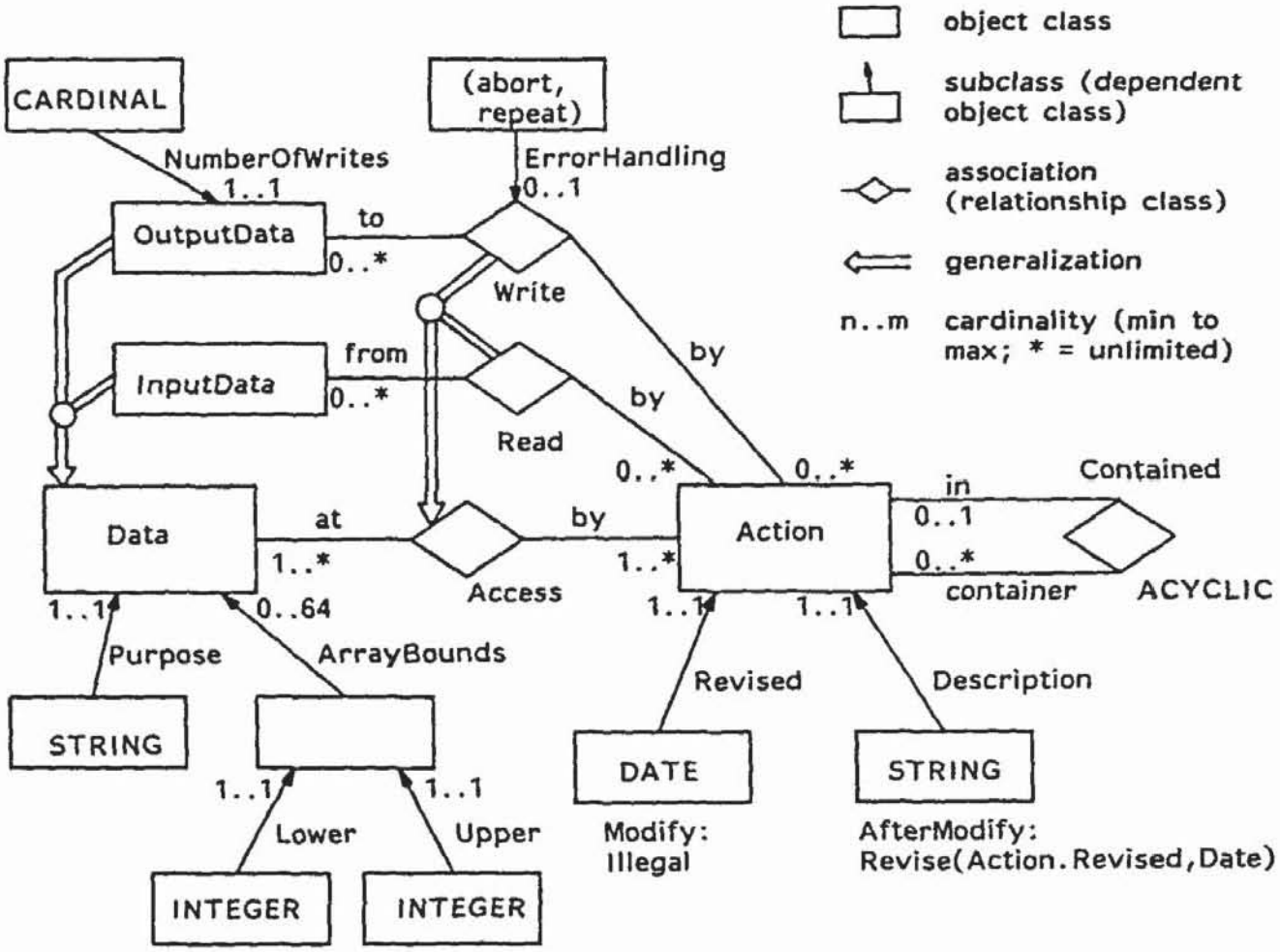

Figure 1: Sample SEED Schema

Explanation: 'Data' is a hierarchically structured object class with classes 'Data.Purpose' and 'Data.ArrayBounds' as subclasses. The latter has again two subclasses, 'Data.ArrayBounds.Lower' and 'Data.ArrayBounds.Upper'. Class 'Data.Purpose' may have objects of type STRING as instances. 'Data.ArrayBounds' has the cardinality 0..64, specifying that any object of class 'Data' may have from zero up to 64 objects of class 'Data.ArrayBounds'.

Classes 'Data' and 'Action' are related by an association 'Access' with cardinalities 1..* and 0..*. '1..*' means that any instance of 'Data' should have at least one 'Access'-link with an instance of 'Action'; there is no upper bound for the number of such links. The roles 'at' and 'by' express that accesses take place at 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'.

We may define an object of class 'Data' by giving its name only, e.g. 'alarms', omitting any subobject (instance of subclass) definition. Such objects may be added later, when more is known about 'alarms'. However, the cardinality 1..1 of class 'Data.Purpose' says that we must eventually supply a string giving the purpose of alarms, i.e. we can formally detect incompleteness.

Class 'Data' is specialized to classes 'InputData' and 'OutputData'. Association 'Access' has specializations 'Read' and 'Write', or conversely: associations 'Read' and 'Write' are generalized to 'Access'. This allows to store a vague information like "The object 'alarms' is a data object which is accessed by the action 'sensor'" as well as a precise information like "'alarms' is an output written twice by 'sensor', and writing is repeated in case of error".

More generally speaking, any instance of 'OutputData' may have the general environment described by class 'Data' (i.e. subobjects of classes 'Data. Purpose' and 'Data.ArrayBounds') plus the special environment described by 'OutputData' (i.e. a subobject of class 'OutputData.NumberOfWrites' and relationships of kind 'Write').

Suppose, an object 'Sensor.Description' of class 'Action.Description' is to be added to the database. Prior to the update execution, the following consistency rules are derived from the schema: (1) There must exist an action with name 'Sensor', (2) there must not already exist an object 'Sensor.Description' (because the max cardinality is 1), (3) the value of the object must be of type STRING, (4) after the update, the procedure Revise(Action.Revised,Date) must be executed. The latter procedure may contain code generating the actual date as an object of class 'Action.Revised'. So every update of an action description automatically updates the revision date, too. On the other hand, procedure 'Illegal' attached to 'Action. Revised' prohibits any direct update of objects of this class. Thus, procedures attached to elements in the schema provide additional semantics that are checked when updates are made.

PATTERNS - A POWERFUL NEW CONCEPT

We can mark any data in the database to be a pattern. Such 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 the inherit operation is as follows: all retrieval operations view patterns as macros, and inherit-operations as macro expansions. However, instead of a real expansion we establish a special inherits-relationship between a pattern and any of its inheritors. Thus pattern information cannot be updated in the environment of the inheritors, but only in the pattern itself. Conversely, any update of a pattern automatically propagates to all inheritors of that pattern.

These semantics are extremely useful for managing variants: A family of variants, consisting of a set of objects and links that is divided into a fixed and several variant parts (one for each variant) can be modelled as follows: Fixed and variant parts are described by normal items. The connections between the fixed and the variant parts are established by patterns such that all variants inherit the same links to the fixed part.

In fig. 2, the fixed part, consisting of objects A and B is connected to a pattern object PO by pattern links PL1 and PL2, respectively. Both variants inherit this pattern. Thus, they both have (inherited) links to objects A and B, i.e. they have the fixed part in common.

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

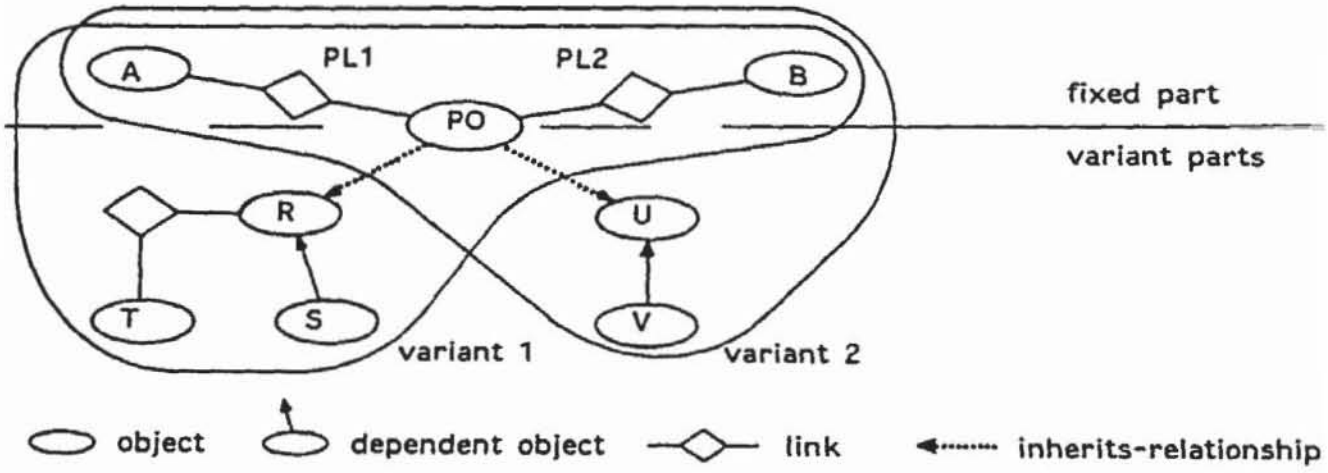

Figure 2: Defining variants by means of patterns

CONCLUSIONS AND PLANS

We have defined a data model that makes conceptual modelling for software engineering easy as it provides features to handle complex structures, vague and incomplete information, consistency, variants, and versions that are vitally needed in this field. The realization of the new features leeds to powerful extensions of the entity-relationship approach.

We are now investigating the problem of variant/version management and the operational aspects of the database more in detail. By 1985, we plan to incorporate the database into a prototype software engineering environment to show the feasibility and usefulness of our ideas in practice.

REFERENCES

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

Dittrich, K.R., A.M. Kotz, J.A. Mülle, P.C. Lockemann (1984). Datenbankkonzepte für Ingenieuranwendungen: eine Uebersicht über den Stand der Entwicklung. 14. Jahrestagung der GI, Braunschweig 1984.

Glinz, M. and J. Ludewig (1984). SEED - Das Datenbanksystem für die Software-Entwicklungsumgebung SEEME. Brown Boveri Research Center, Research Report No. KLR 84-143 C.

Osterweil, L. (1981). Software environment research: directions for the next five years. IEEE Computer 14,4. 35-43.

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

Stenning, V. (ed.) (1983). Requirements for software engineering databases. Imperial Software Technology Ltd., London.