The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes*

The GOODSTEP Team

Abstract

The goal of the GOODSTEP project is to enhance and improve the functionality of a fully object-oriented database management system to yield a platform suited for applications such as Software Development Environments (SDEs). The baseline of the project is the O_2 database management system (DBMS). The O_2 DBMS already includes many of the features required by SDEs. The project has identifed enhancements to O_2 in order to make it a real software engineering database management system.
These enhancements are essentially upgrades of the existing O_2 functionality, and hence require relatively easy extensions to the O_2 system. They have been developed in the early stages of the project and are now exploited and validated by a number of software engineering tools built on top of the enhanced O_2 database system. To ease tool construction, the GOODSTEP platform encompasses tool generation capabilities which allow for generation of integrated graphical and textual tools from high-level specifications. In addition, the GOODSTEP platform provides a software process toolset which enables modeling, analysis and enaction of software processes and is also built on top of the extended O_2 database. The GOODSTEP platform will be validated using two CASE studies carried out to develop an airline application and a business application.

*This work has been funded by the EU under contract No. 6115 (ESPRIT-III project GOODSTEP)

1 Objective of GOODSTEP

The goal of the GOODSTEP project is to develop a sophisticated database system dedicated to the support of software development environments (SDEs) and make the basis for a platform for SDE construction with a software process toolset and generators for graphical and textual integrated tools implemented on top of it.

The GOODSTEP project started September 1992 and will last for three years. This paper mainly reports on the first year of work within the project.

The baseline of the project is an existing European commercially available object-oriented database product: O_2 [4]. Rather than developing a new database management system from scratch, GOODSTEP will enhance and improve this product.

Besides the enhancements and improvements of O_2 which make it an admirably suited system for SDEs, the project provides a number of test cases and performs case studies to evaluate and justify the approach. This includes porting and developing a number of existing software engineering tools on top of the new platform, the development of tool generation capabilities to exploit the features of a fully object-oriented system, and the development of advanced software process modeling capabilities which, once again, exploit the provided features of O_2.

The choice of an object-oriented database management system as baseline of the project derives from the inadequacy of relational database systems for software engineering applications, which has been recognized for quite some time [22, 25]. This has resulted in a great effort in both the indus-

trial and research communities to extend the current database technology towards more powerful and flexible database management systems. In particular, in this context we are interested in the work done on the development of object-oriented database systems.

The class of object-oriented database systems can be roughly classified in two categories: those offering the full functionalities of the object-oriented data model, which we will refer to from now on as *object database systems* (ODBSs) [10], and those offering only a subset of the object-oriented model, which we will call *structurally* object-oriented database systems. The main distinction between structurally object-oriented databases and object database systems, which is also the main drawback of the first class, is that most of the structurally object-oriented database systems, such as PCTE/OMS [17], Damokles [12], assume a certain level of granularity of the objects to be stored and retrieved and that they further cannot define encapsulation of data structures by operations. These systems either support relations between coarse-grained objects such as products of the SE life cycle (e.g. A is the specification of B, A is owned by developer Q, etc.), or else support a rather fine-grained level of objects such as syntactic units of programs or specifications (i.e. statements, variables, procedures, etc.).

In practice, many software engineering tools require support for both levels of granularity. By contrast, object database systems are the best approach to support sophisticated efficient storage and retrieval of objects at arbitrary levels of granularity [13].

The project amply demonstrates two important features of an object database system. In the first place, *it enables much easier implementations of SE-tools than when using conventional DBMSs.* This is because the richer semantics of the object oriented model is most suited to the complex data handled in software engineering applications, and also because there is no problem of fixed granularity of objects. In the second place, *it significantly improves the functionality of software tools which can be based upon it.*

The software architecture of GOODSTEP is illustrated in Figure 1. GOODSTEP enhances the functionalities offered by O_2, by adding new functionalities to its kernel or on top of it. Moreover, two tool sets have been developed together with the enhanced O_2 system to constitute the GOODSTEP platform. The first toolset (TCT) supports the generation of new tools. The other set of tools support development, simulation and execution of software processes (SPT). In particular it integrates the tools generated by the first tool set. Both SPT and TCT support the development of a customized SDE.

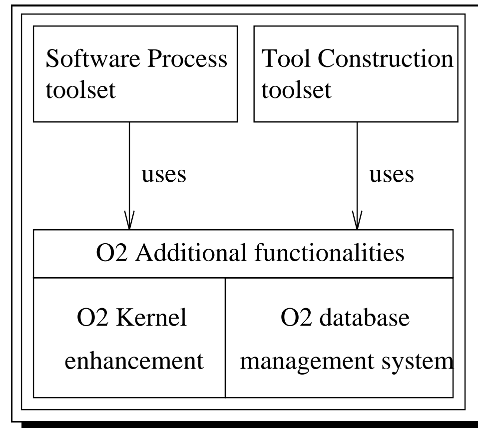

Figure 1: The GOODSTEP SDE platform

The rest of the paper is structured as follows: In Section 2, we first describe the GOODSTEP platform in some detail. In Section 3, we describe the fundamental requirements for the underlying database system as required by the GOODSTEP platform. In Section 4, we show how these requirements are addressed by O_2 and we describe the planned O_2 extensions. Finally, Section 5 concludes the paper while drawing attention on ongoing and further work.

2 Building the GOODSTEP Platform for Software Development Environments

The methods and languages to be used for the development of a software application – be they graphical or textual languages – depend on the application domain. For example, real-time applications require different specification and implementation languages

than financial or medical applications. Moreover, the process models that support an application development best can not be predefined. They depend not only on the application domain, but also on the organization and on the people who are running the processes. Current SDEs suffer from their selection of specific combinations of languages and often assume a particular process. Both usually do not completely cover the needs of any software development and impose a pre-specified development mode. Instead, the software industry demands customizable SDEs, which may be easily adapted to the evolving needs of software development. GOODSTEP addresses this need by providing means to

- define the process used for developing a software system within the software process tool-set and
- generate the tools needed during the course of a software process using the tool construction tool-set.

Using the process tool-set a model tailored to the needs of a particular institution or even project can be modeled, analyzed and later used for running a software project. Using the tool construction tool-set, it will be possible to define and generate conceptual schemas and define and generate a set of integrated syntax-directed software development tools from appropriate specification languages.

2.1 Software Process Modeling and Enactment

Software process modeling and enactment is supported in GOODSTEP by the SPADE (Software Process Analysis Design and Enactment) environment. SPADE provides a domain-specific language for the modeling and enactment of software processes called SLANG (Spade LANGuage). SLANG is based on high-level nets and is given formal semantics in terms of a translation scheme from SLANG objects into ER nets. ER nets [18] are a mathematically defined class of high-level Petri nets that provide the designer with powerful means to describe concurrent and real-time systems. In ER nets, it is possible to assign values to tokens and predicates to transitions, describing the constraints on tokens consumed and produced by transition firings.

2.1.1 SLANG

We describe SLANG by means of a simple example. A SLANG specification of a process fragment is presented in Figure 2. For an elaborated discussion of SLANG, we refer to [7].

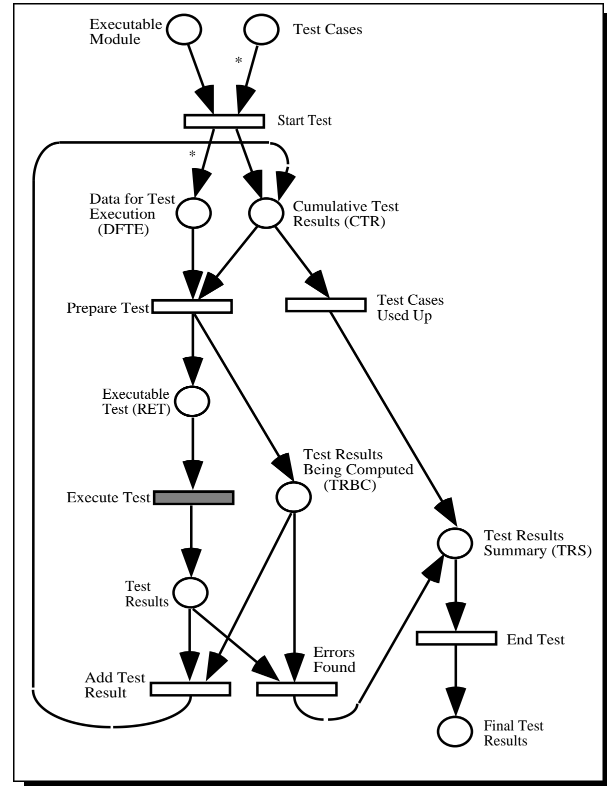

Figure 2: A Model of a Process Fragment in SLANG

The example models a process *activity* to test executable modules of a software system. The net associated with the activity is activated as soon as another process activity (SLANG nets are hierarchically organized in activities) puts a token that represents a module successfully compiled on the place *Executable Modules* and some other tokens representing test data on the place *Test Cases*. For each such test case, a test is executed (modeled by transition *Execute Test*)

and the results are cumulatively recorded in an error report. If no more test cases are available, transition *EndTest* fires, thus ending the activity, and producing a token that models the error report in place *Final Test Results*. Note that *ExecuteTest* is described as a black transition; meaning that the transition is not atomic; it invokes the execution of the testing program and then the execution of the net is resumed without waiting the tests to be completed. This mechanism is used in SLANG to model the invocation of tools such as those generated with the tool construction toolset.

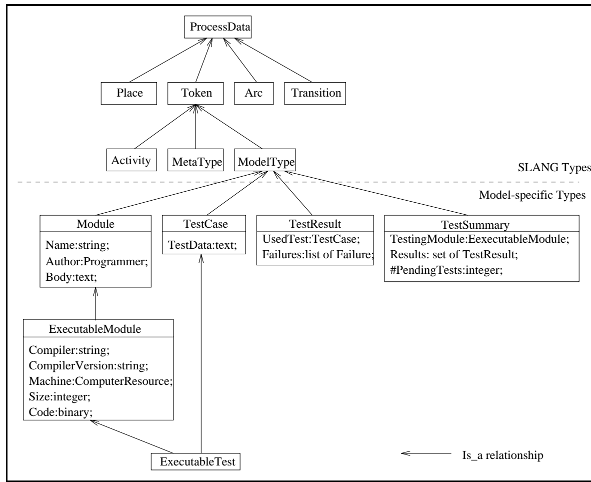

Figure 3: Type Definition of Tokens for Net in Figure 2

To complete the formal definition of the net, the token types must be defined. These types are defined by attaching type definitions to places. Type definitions are organized in a type hierarchy, defined by an "is-a" relationship. The root of the hierarchy is the type `ProcessData`. Types inherit the attributes and operations of their ancestors in an object-oriented style. The example of the type hierarchy used in the net of Figure 2 is given in Figure 3.

To support process evolution, i.e. "changes on the fly" [21], type descriptions may be added or changed during process enactment. The change of a type requires instances of that type to change accordingly.

2.1.2 SPADE

SPADE [6]is a software process environment that supports the enactment of process models written in SLANG. The enactment of the process model causes the automatic execution of computer-based actions and guides the behavior of people involved in the process. In addition to the description of a software production process, a SLANG process model may also include the specification of a software meta-process. Therefore, process enactment involves the execution of activities of both the production process and the meta-process. The meta-process models those actions that do not aim at software production, but concern the management of the process itself: creation/modification of activities, object types, etc. The reflective nature of SLANG makes it possible to manipulate the process definition (i.e., *ProcessTypes* and *ProcessActivities*) in the same ways other process data are manipulated, therefore enabling process evolution.

Important requirements for database systems arise when we consider the problem of process evolution [5]. All information describing a SLANG specification (process type descriptors and process activities) and the process state (all instances of *ProcessData* and its subtypes) have to be stored in a repository, i.e an O_2 database. The SLANG interpreter uses O_2 to access both the description of the process model and the process data produced and modified as result of its enactment.

While it is not possible to change the definition of the SLANG language (fixed part of the schema), all other components can be changed. Changes to the instances of the fixed part of the schema (e.g. arcs and transitions), and changes to the modifiable part of the schema (e.g. token types) correspond to changes in the process model. Changes to the instances of the modifiable part correspon to changes in the state of the enacted process model.

2.2 Tool Construction

The GOODSTEP platform contains two tool generators. The first one is the GraphProject compiler that is capable of gener-

ating graphical tools. The second is the GENESIS compiler which takes a specification written in the object-oriented tool specification language GTSL [8] and automatically derives textual syntax-directed tools. We first discuss the functional and non-functional properties users[1] require from generated tools and then sketch some issues on tool design which later on have an impact on the database functionality required to make tool generation feasible.

2.2.1 Requirements on Software Development Tools

The tools to be generated will be used by users to edit, analyze and transform documents. To give as much support to users as possible, the tools must be *syntax-directed* according to the languages in which documents are written.

Besides dealing with errors concerning the context-free syntax, tools should also deal with errors concerning both the *internal static semantics* of documents and *inter-document consistency constraints.* The tools may allow temporary inconsistencies to be created during edit operations, since document creation in a way which avoids such temporary inconsistencies is impractical in many cases.

Obviously, documents must be *stored persistently* because they must survive editing processes. Moreover, users require tools to operate as safely as possible, i.e. in case of a hard- or software failure they expect *integrity preservation* of documents (their immediate usability by the same or other tools) against hard- or software failures. Also significant user effort must not be lost in case of a failure, i.e. any completed change a user performed to a document must persist in the database.

When the consequences of user actions are persistent, users must have the ability to backtrack or "undo" such actions when mistakes are made. Thus, users may want to store intermediate *versions* of a document so that they can revert to a previous revision if their subsequent modifications turn out to be ill-chosen.

Finally, users expect acceptable *performance* from the tools such that their work-flow is not interrupted.

[1] As the users of the GOODSTEP platform have different roles, we distinguish in the sequel users which are the developers that use the customized GOODSTEP platform in order to develop an application from SDE builders who use the features provided by the GOODSTEP platform for customizing it to obtain a particular SDE.

2.2.2 Issues in Tool Design

What is a document in the database? The common internal representation for syntax-directed tools such as syntax-directed editors, analyzers, pretty-printers and compilers is a syntax-tree of some form. In practice, this abstract syntax-tree representation of documents is generalized with *context-sensitive edges* to an abstract syntax-graph for reasons such as efficient execution of documents, consistency preservation of documents, and user-defined relations within documents. Such context-sensitive edges are not confined to within individual documents – context sensitive edges frequently exist between components of distinct documents. As an example c.f. Fig. 4 where they relate nodes in a graph of module interface specifications with corresponding nodes in another graph of module body specifications. This leads to a *project-wide abstract syntax graph.* For a detailed discussion of context-sensitive edges we refer to [14].

How is it stored? Due to the requirements of persistence and integrity, a persistent representation of each document under manipulation must be updated as soon as each user-action is finished. Typically a user-action affects only a very small portion of the document concerned, if any. Given that the representation under manipulation is an abstract syntax-graph, the update can easily become inefficient if, firstly, a complex-transformation between the graph and its persistent representation is required and, secondly, the persistent representation is such that large parts of it have to be rewritten each time, although not being modified. This would for instance be the case, if we had chosen to store the graph in a sequential operating system file which is updated at the end of each user-action.

Such inefficiency can be avoided completely

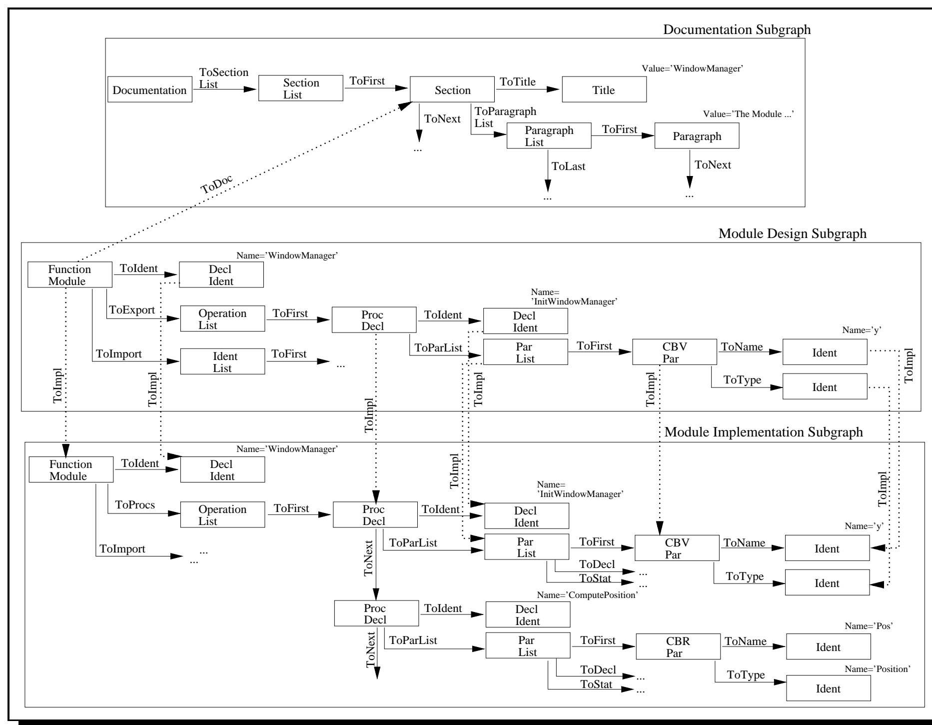

Figure 4: Excerpt of a project-wide Abstract Syntax Graph

if the persistent representation takes the form of an abstract syntax graph itself, with components and update operations that are one-to-one with those required by the tools concerned.

3 Requirements on Databases for Software Engineering

In this section we outline the requirements on a DBMS that arose during design of both the software process toolset and the tool construction tool set. A detailed discussion of the requirements can be found in [14].

DDL/DML The types of tokens as defined by SLANG types must be established in the schema in order to have the process engine storing tokens in O_2. For the same reason, the overall structure of project-wide abstract syntax graphs has to be defined in terms of the data definition language of the database system and it must be established and controlled by the database's conceptual scheme. This implies that the data definition language is appropriate to cope with the complexity inherent in project-wide syntax-graphs. To manage this complexity the distinction of objects and types, encapsulation of objects' attributes by operations and information-hiding as well as inheritance to express generalisation/specialisation should be applicable to the data definition. Therefore, a powerful type mechanism including object constructors for expressing different types of aggregation and method definitions to achieve encapsulation must be provided.

Schema Updates Due to the reflective nature of SLANG we require support for schema updates from the database system as a prerequisite for a SPADE implementation. This has a number of implications. First of all, the database system must enable introduction of new type definitions, changes to existing type definitions and even deletion of type definitions even though the database may have been filled with instances already. As the process engine cannot be stopped for executing the schema updates, these updates must be possible, even if other concurrent

DBSE applications such as tools are operating against the database. Process data may have been created and accessed under control of the schema which has to undergo a change. These data must not be lost or corrupted otherwise by the change. This requires means to adapt the data structures of existing data to the changed structures. Since the process engine can not be stopped for performing these changes, migration of data must be atomic, i.e. either done completely or not at all.

Versions The DBSE must support creation and management of versions of those subgraphs that represent versionable documents. In particular, it must enable its clients to derive a new version of a given subgraph, to maintain a version history for a subgraph, to remove a version of a given subgraph, and to select a current version from the version history.

Views A project-wide abstract syntax graph may contain a lot of redundant information – in Fig. 4 nodes of type *FunctionModule*, *DeclIdent*, *OperationList* and the edges *ToIdent* and *ToExport* are duplicated in the interface and body subgraphs. Eliminating this duplication by sharing the aggregation subtrees concerned has the following advantages: (1) the conceptual schema is simplified (2) storage of the schema and corresponding data requires less space, and (3) consistency preservation especially across document boundaries becomes much easier. If subtree sharing is to be used, tools accessing the project graph need a view mechanism like that offered in many relational database systems, to maintain appropriate separation of tool concerns and so allow separate tool development and maintenance. These tool-oriented views must be regarded as virtual graphs since they are not actually stored in the DBSE, but updates on views by tools must propagate automatically to the underlying project graph.

Active capabilities When abstract syntax-graphs are changed by one user, other users may have to be notified about this change. This can be achieved with a trigger facility. Triggers may be used also to support control integration of the tools and process integration. Control integration implies that a tool can invoke another tool to perform some piece of software process it is carrying out. A process model governs the invocation of tools and a tool is invokable only if the model allows it. Triggers are required also to support the integration of software process interpreters and the integration between these interpreters and the tools.

Transactions The database system must preserve the integrity of the abstract syntax-graph and the process state against hard- or software failures. Therefore the system must support atomic transactions, i.e., a transaction mechanism that enables grouping a sequence of update-operations such that they are either performed completely or not performed at all. To ensure that a tool can recover in case of a failure to the state of the last completed user-action, each completed DBSE transaction must be durable.

Distribution In an SDE distribution of activities is important. This can be achieved by allowing for distributed accesses from the users' (client) workstations to a database server. Following this approach, however, care must be taken that the server does not become a performance bottleneck for the whole environment.

4 Enhancing the O_2 ODBS

Relational database systems are inappropriate for storing project graphs, since (1) the data model can not express syntax graphs appropriately, (2) relational database systems do not support versioning of document subgraphs and (3) relational views are not updatable in general. No structurally object-oriented DBMS meets all of our requirements. Those that are capable of efficiently managing project graphs such as GRAS [20], lack functionality w.r.t. views, versioning, access rights and adjustable transaction mechanisms, and distribution. Others that offer these functionalities such as PCTE/OMS [17] or CAIS-A [2] are unable to efficiently manage the large

collections of small objects as they occur in project graphs and are therefore inappropriate. The approach of GOODSTEP is therefore to start from an existing object-oriented DBMS which is the O_2 system since this already addresses some of the requirements presented previously. This will be discussed in Subsection 4.1. A major effort in GOODSTEP is devoted to enhancing O_2 enhancing or adding functionalities for schema evolution, version management, view definitions and active database capabilities. This will be addressed in Subsections 4.2-4.5.

4.1 Features offered already by O_2

In [13] we have shown how to implement the structure definition of project-wide ASGs within a schema of a fully object-oriented DBMS in general and within O_2 in particular. In O_2 we would therefore define common properties of nodes within *classes*. The *type* of the class then defines attributes and outgoing edges as *instance variables*. Navigation along edges is implemented by dereferencing. Consistency of the graph definition can be checked at compile-time based on O_2's *type system*. For schema simplification purposes, *inheritance* is used to define common properties of nodes only once in a superclass. Integrity of a syntax graph is enforced by *encapsulation*, i.e a tool can not modify edges and attributes directly, but must use the methods defined for that purpose. The computations necessary for performing these methods can be done within the schema because the data definition language of O_2 is *computational complete*. We have shown in [5] how the execution of the process engine can be implemented using the O_2 *query language* [3].

O_2 offers a *transaction mechanism* which can be used for grouping a set of update statements such that they are performed completely and are then durable or not at all. This enables updates of process states as well as changes to a syntax-graph to be done in an integrity preserving manner.

O_2 provides for *distributed access* from several clients to several central databases. These client/server accesses exchange pages. Given the clustering mechanism of O_2 [9] we can manage to transfer many nodes of a syntax graph with a single network call. A further contribution to efficiency is that execution of methods defined in the schema is done on the client thus avoiding a performance bottleneck on the servers.

4.2 Schema Evolution

Schema evolution in an object database system refers to the ability to change both the schema and consequently the database. Every time a schema is modified the database has to be updated to be brought to a consistent state with respect to the new schema. A schema can be changed using special primitives, see for example [24]. The corresponding database updates are performed using *user-defined conversion functions* whose input parameters are instances of the old and the new schema class definitions. Being executed they transform objects of the database to conform to the new schema. The SDE builder has to define a conversion function for each modified class in the new schema. System default transformations are applied in case no explicit conversion functions are given by the builder.

From a point of view of an SDE builder, after execution of the appropriate conversion functions, the entire database conforms to the the new schema. From an implementation point of view, conversion functions are updates to the database. There are mainly two strategies for implementing database conversion functions: immediate and lazy [19]. In the first case, all objects of the database are updated immediately after the execution of all conversion functions. In the second case, objects are updated only when used (i.e. conversion functions are executed only when objects are effectively used).

No matter what strategy is used, the effect on the database has to be the same: the database must be in a consistent state with respect to the new schema [16]. For a more detailed discussion of O_2 schema updates using lazy evaluation of conversion functions we refer the interested reader to [15].

4.3 Versions

Our goal for extending the O_2 database system with version management facilities is

to provide a Version Manager, implemented as a predefined O_2 class which provides a set of methods for manipulating the different versions.

The granularity of versioning is that of composite objects. These composite objects implement subgraphs of the project-wide abstract syntax graph. We provide a class `Version` which enables a tool to define composite objects at run-time. Instances of this class are the smallest entity known by the version manager. The different objects belonging to a composite object are versioned together. Therefore the class `Version` provides several basic methods which allows the SDE builder to:

- Create a version (i.e. initialize a version history graph),
- Add or delete objects to a version, thus including/removing them in/from the versioned composite object
- Derive a new version from an existing version,
- Determine a particular version as default version,
- Select a particular version different from the default version,
- Delete a version by removing it from the DAG,
- Retrieve a version,
- Navigate through the version history graph.

This will provide a tool builder with the basic functionality to maintain different versions of those subgraphs of a project-wide abstract syntax graph that represent versioned documents.

4.4 Views

The baseline of this part of the project is the view mechanism defined in [1] and extended in [23]. In our approach, the definition of a view is similar to the definition of a schema. A *virtual schema* is, like a normal schema, an organizational unit meant to encapsulate a set of related intentional definitions. The main difference is that a real schema describes the structure and behavior of real data stored in the database whereas a virtual schema describes a virtual world.

A view is thus in our context a special kind of schema with certain restrictions. We use the term *virtual schema* as a synonym of view in what follows and the term *real schema* is used in contrast to virtual schema to avoid confusion.

Like a real schema, a view includes definitions of classes, methods, types, functions and named objects. It may also import and export definitions from other schemas, although these mechanisms are given a slightly different semantics. In addition, virtual definitions can be included in a virtual schema. As a matter of fact, the distinguishing point between a real and a virtual schema is the fact that the latter has at least one virtual or imaginary class (possibly imported) whereas the former has only real classes.

A virtual schema is always derived from another virtual or real schema, which we call its *root schema*. When we define a view we must therefore declare its root schema. The root schema of a view determines on which databases it can be activated, namely those databases instantiated from its root schema.

Using these virtual schemas an SDE builder is enabled to define virtual abstract syntax graphs on top of conceptual graphs stored in the database. These graphs can be accessed and modified in a way customized towards particular tools.

4.5 Active Capabilities

In the framework of GOODSTEP active rules have been introduced in O_2 as a means of supporting SDE construction. We can only sketch here and refer for an elaborated discussion to [11]. Rules we consider are production rules based on the Event-Condition-Action (ECA) formalism. The overall semantics of an ECA rule is: "Whenever the event E occurs, if the Condition C holds then execute Action A".

In our context, rules are components of an O_2 schema; they are defined at the same level as types, classes and applications. They are thus considered at a higher level than programs, methods and data manipulation. This approach allows to control the execution of more general operations than meth-

ods calls or manipulation of a single entity (an entity is an object or a value) and to associate rules to transaction, program or application executions. Rules respect encapsulation and transparency: they are triggered by authorized operations related to persistent or transient entities. Rules can be exported/imported between schemas which increases reusability. Finally, for the purpose of programming guidelines, rules are isolated from programs and methods: a rule can be activated or deactivated only through rules.

The event part of a rule definition specifies which events will trigger the rule. Proposed event types can be divided into two categories. The first category is made of entity manipulation event types which are generated by manipulations (creation, deletion, update, etc.) of entities. The second category is made of applicative event types which are associated to the begin or end of a transaction, an O_2 program or application. The Condition part is made up of predicates (O_2SQL queries) over entities. The Action part is made of any O_2C code. Conditions and Actions can operate on persistent or transient entities.

Based on the atomic transaction model of O_2, we defined two kinds of rules: (1) immediate rules which are executed right after the occurence of the triggering event and, (2) deferred rules (corresponding to cumulative changes on entities) which are executed at the end of the transaction (just before its validation) in which triggering events occured. The choice of rules to be executed is based on (i) a total rule ordering ensured by the system and (ii) on the notion of execution cycle. A cycle describes the execution of a sequence of operations belonging to user-defined transaction, a program or a rule. Every execution cyle is associated with a *delta structure* containing data related to the triggering operations considering the net effect of these operations. *Delta structures* can be accessed in Conditions and Actions using specific operators.

5 Conclusions and Ongoing Work

In this paper, we have limited the presentation of the project mainly to show the rationale of the project. We have discussed the requirements posed to an SDE and the way we propose to tackle them. The first integrated version of the GOODSTEP platform is due at the end of 1994.

Currently, we are starting to use the GOODSTEP platform within two case studies. The aim of the first case study is to customize the platform to an SDE for use within typical information system development processes. This SDE will then be used by an industrial partner for development of an information system supporting university adminstration. In a second case study we are going to customize the GOODSTEP platform for use within airline software projects of another industrial partner. These projects reuse C++ classes from a variety of class libraries. The customised SDE for this project is going to support the development and maintenance process of C++ class libraries.

Further Information and Acknowledgements

Information about GOODSTEP including a periodically updated series of technical reports can be obtained via anonymous ftp from `ftp.inria.fr` in `/pub/INRIA/Projects/Verso/GoodStep-Library`. Inquiries about the project will be answered by `zicari@dbis.informatik.uni-frankfurt.de`.

The members of the GOODSTEP team and the institutions supporting GOODSTEP are: S. Abiteboul (INRIA), M. Adiba (Uni Grenoble), J. Arlow (British Airways), P. Armenise (Engineering), S. Bandinelli (Cefriel), L. Baresi (Cefriel), P. Breche (Uni Frankfurt), F. Buddrus (Uni Frankfurt), C. Collet (Uni Grenoble), P. Corte (Engineering), T. Coupaye (Uni Grenoble), C. Delobel (INRIA), W. Emmerich (Uni Dortmund), G. Ferran (O_2 Technology), F. Ferrandina (Uni Frankfurt), A. Fuggetta (Cefriel), C. Ghezzi (Cefriel), S. E. Lautemann (Uni Frankfurt), L. Lavazza (Cefriel), J. Madec (O_2 Technology), M. Phoenix (British Airways), S. Sachweh (Uni Dortmund), W. Schäfer (Uni Dortmund), C. Souza Dos Santos (INRIA), G. Tigg (British Airways) and R. Zicari (Uni Frankfurt).

References

[1] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. of the ACM SIGMOD Conf. on Management of Data, Denver, Co*, pages 238–247. ACM Press, 1991.

[2] Ada Joint Program Office. Common Ada Programming Support Environment (APSE) Interface Set (CAIS), Revision A. Technical Report DoD-STD-1838A, U.S. Department of Defense, 1988.

[3] F. Bancilhon, S. Cluet, and C. Delobel. Query languages for object-oriented database systems: the O_2 proposal. In *Proc. DBPL, Salishan Lodge, Oregon*, June 1989.

[4] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: the Story of* O_2. Morgan Kaufmann, 1992.

[5] S. Bandinelli, L. Baresi, A. Fuggetta, and L. Lavazza. Requirements and Early Experiences in the Implementation of the SPADE Repository using Object-Oriented Technology. In S. Nishio and A. Yonezawa, editors, *Proc. of the First JSSST International Symposium Kanazawa, Japan*, volume 742 of *Lecture Notes in Computer Science*, pages 511–528. Springer, 1993.

[6] S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza. The Architecture of the SPADE-1 Process-Centered SEE. In B. C. Warboys, editor, *Software Process Technology — Proc. of the 3rd European Workshop, EWSPT '94, Villard de Lans*, volume 772 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 1994.

[7] S. Bandinelli, A. Fuggetta, and C. Ghezzi. Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, 19(2):1128–1144, 1993.

[8] W. Beckmann, J. Brunsmann, D. Dong, W. Emmerich, P. Kroha, W. Reimer, S. Sachweh, and W. Schäfer. The Goodstep Tool Specification Language Reference Manual. Deliverable ESPRIT Project GOODSTEP 6115-6P, Commission of the European Communities, November 1993.

[9] V. Benzaken, C. Delobel, and G. Harrus. Clustering Strategies in O_2: An Overview. In *[4]*, pages 385–410. 1992.

[10] R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufman, 1993.

[11] C. Collet, T. Coupaye, and T. Svensen. NAOS Efficient and modular reactive capabilities in an Object-Oriented Database System. In *Proc. of the* 20th *Int. Conf. on Very Large Databases, Santiago, Chile*, 1994. To appear.

[12] K. R. Dittrich, W. Gotthard, and P. C. Lockemann. Damokles – a database system for software engineering environments. In R. Conradi, T. M. Didriksen, and D. H. Wanvik, editors, *Proc. of an Int. Workshop on Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 353–371. Springer, 1986.

[13] W. Emmerich, P. Kroha, and W. Schäfer. Object-oriented Database Management Systems for Construction of CASE Environments. In V. Mařik, J. Lažanksý, and R. R. Wagner, editors, *Database and Expert Systems Applications — Proc. of the* 4th *Int. Conf. DEXA '93, Prague, Czech Republic*, volume 720 of *Lecture Notes in Computer Science*, pages 631–642. Springer, 1993.

[14] W. Emmerich, W. Schäfer, and J. Welsh. Databases for Software Engineering Environments — The Goal has not yet been attained. In I. Sommerville and M. Paul, editors, *Software Engineering ESEC '93 — Proc. of the* 4th *European Software Engineering Conference, Garmisch-Partenkirchen, Germany*, volume 717 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 1993.

[15] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proc. of the 20th Int. Conference on Very Large Databases, Santiago, Chile*, 1994. To appear.

[16] F. Ferrandina and R. Zicari. Object Database Schema Evolution: are Lazy Updates always Equivalent to Immediate Updates? In *Proc. of the OOPSLA Workshop on Supporting the Evolution of Class Definitions, Washington, DC.* ACM Press, 1993.

[17] F. Gallo, R. Minot, and I. Thomas. The object management system of PCTE as a software engineering database management system. *ACM SIGPLAN NOTICES*, 22(1):12–15, 1987.

[18] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzé. A Unified High-level Petri Net Formalism for Time-critical Systems. *IEEE Transactions on Software Engineering*, 17(2):160–173, 1991.

[19] G. Harrus, F. Velez, and R. Zicari. Implementing schema updates in an object-oriented database system: a cost analysis. Technical report, GIP Altair, 1990.

[20] C. Lewerentz and A. Schürr. GRAS, a management system for graph-like documents. In *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases.* Morgan Kaufmann, 1988.

[21] N. H. Madhavji and W. Schäfer. Prism – Methodology and Process-Oriented Environment. *IEEE Transactions on Software Engineering*, 17(12):1270–1283, 1991.

[22] D. Maier. Making database systems fast enough for CAD applications. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 573–582. Addison-Wesley, 1989.

[23] C. Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In *Proc. of the 4th Int. Conf. on Extending Database Technology, Cambridge, UK*, Lecture Notes in Computer Science. Springer, 1994. To appear.

[24] R. Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In *[4]*, pages 146–182. 1992.

[25] R. Zicari and C. Bauzer-Meideros. New Generation Database Systems. In P. Locopoulos and R. Zicari, editors, *Conceptual Modeling, Databases and CASE — An Integrated View of Information Systems Development.* John Wiley, 1992.