

Requirements and Early Experiences in the Implementation of the SPADE Repository using Object-Oriented Technology *

Sergio Bandinelli[†] Luciano Baresi Alfonso Fuggetta Luigi Lavazza

CEFRIEL - Politecnico di Milano

Via Emanuelli 15, 20126 Milano (Italy)
Tel.: +39-2-66100083, Fax: +39-2-66100448
e-mail contact: lavazza@mailier.cefriel.it

Abstract

Software development environments (SDEs) pose pressing requirements to the supporting repositories. This paper describes these requirements, as derived within the SPADE project. SPADE is a process centered environment being developed at CEFRIEL and Politecnico di Milano. Aim of the paper is to report the experiences that the authors have gained in building a repository for SPADE using O_2 , a “state of the art” object-oriented DBMS.

1 Introduction

In recent years, many researchers in the software engineering area (see for example [13, 16]) have identified the software process as the key issue to obtain higher quality products, improved productivity, more controllable projects. By software process we mean the set of activities, rules, methodologies, tools, and roles that participate in the development of software within a given organization.

There is no single process that can be used by any organization, for any kind of product, any development environment, or any software lifecycle. It is necessary to be able to envisage different processes, depending on the characteristics of the product, the market, and the development organization. For this purpose there has been an increasing effort in designing and developing languages and the related support technology to formally describe, assess, and wherever possible, automate software processes. The degree of automatism of the process enactment¹ depends on

*This work has been partially supported by ESPRIT project 6115 GoodStep - General Object-Oriented Databases for Software Processes

[†]S. Bandinelli is partly supported by DEC

¹“enactment” means execution, in the software process jargon. This special word has been introduced to stress the concept that the execution is only partly automatic.

the different activities: low-level activities, such as calling a compiler, may be fully automatic, while brain-intensive activities, such as transforming a detailed design into code, are simply guided by the process interpreter, for example producing an agenda for each participant in the development, with reminders of the activities to be still accomplished.

Process centered SDEs (PSDEs) are built around a process interpreter, and in general include a repository for the process data (i.e. the software artifacts plus the process specific information) and a set of integrated tools. Many authors addressed the issue of identifying the requirements for a database system supporting a software engineering environment (see for example the early work by Bernstein [7]). More recently, other works have addressed the issue of the suitability of object-oriented databases as a vehicle to implement software engineering repositories ([11, 2]). Experiences in the usage of object-oriented technology in building software engineering environments (although not process centered) have also been reported [17].

SPADE is a process-centered environment that is built around an OODB. The decision of using an Object-Oriented database [1] for building the repository of SPADE descends from the consideration — supported by [11, 17] among others — that many of the requirements described in [7] have been satisfied — at least partially — by OODBMSs. Other database technologies, like relational databases or structurally object-oriented databases, are clearly inadequate [10, 9]. This activity is partially supported by the ESPRIT project GoodStep, that aims at extending the O_2 OODB to support the development of advanced process-centered software engineering environments [19].

The paper is organized as follows. Section 2 briefly describes the SLANG software process language. Section 3 gives a conceptual description and classification of data involved in the definition and execution of SLANG models. Section 4 describes the requirements for the database that are posed by the peculiar features of the language and of the whole environment. In section 5 the O_2 based implementation is described and discussed. Section 6 draws some conclusions and presents future research directions.

2 The SLANG software process language

SPADE (Software Process Analysis, Design and Enactment) is a software process environment, being jointly developed at CEFRIEL and Politecnico di Milano, that provides mechanisms for the definition, analysis, enactment, and evolution of a software development process.

SPADE provides a domain-specific language for modeling and enacting software processes called SLANG (SPADE LANGuage) [3, 4, 5, 6]. SLANG is based on high-level nets and is formally defined in terms of a translation scheme from SLANG into ER nets. ER nets [12] 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 relations to transitions, describing the constraints on tokens consumed and produced by transition firings.

2.1 SLANG features

A SLANG process specification is a pair of sets:

$$SLANGSpec = (ProcessTypes, ProcessActivities)$$

ProcessTypes is a set of type specifications which define the data manipulated by the process; *ProcessActivities* is a set of activity definitions, which specify the process computations in terms of logical work units using an extended high-level Petri net formalism. These features are now described in more detail.

Process types. In SLANG, all process data are typed, and data with the same properties and characteristics may be grouped as belonging to the same type. 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. An example of such type hierarchy is given in figure 1.

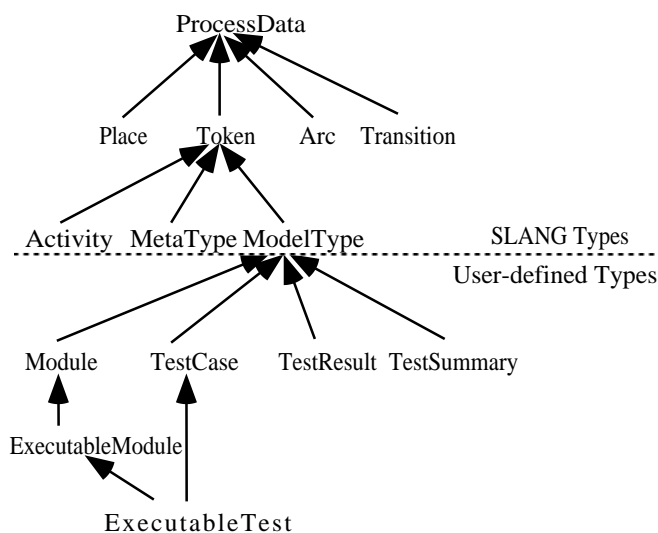


Figure 1: An “is-a” hierarchy of type definitions.

The set of type definitions whose root is “*ModelType*” varies from one SLANG specification to another, depending on the process to be described. Type descriptions may be added or changed during process enactment. The change of a type requires data instances of that type to change accordingly. Change propagation can be specified within the process definition as being either *lazy* (the changes are visible only when the modified type is used), *eager* (the change has an immediate impact on the process state), or any intermediate strategy (partial propagation).

In order to support the ability of specifying evolving processes, SLANG offers reflective mechanisms. During process enactment, type definitions and activity definitions can be manipulated as data. Because all process data must be typed, two special types are therefore introduced: *Activity* and *Metatype*. *Activity* is the type whose instances are activity definitions; *Metatype* is the type whose instances are type definitions.

Process activities. *ProcessActivities* defines a set of activities by means of an extended high-level Petri net formalism, where tokens carry structured information of arbitrary complexity.

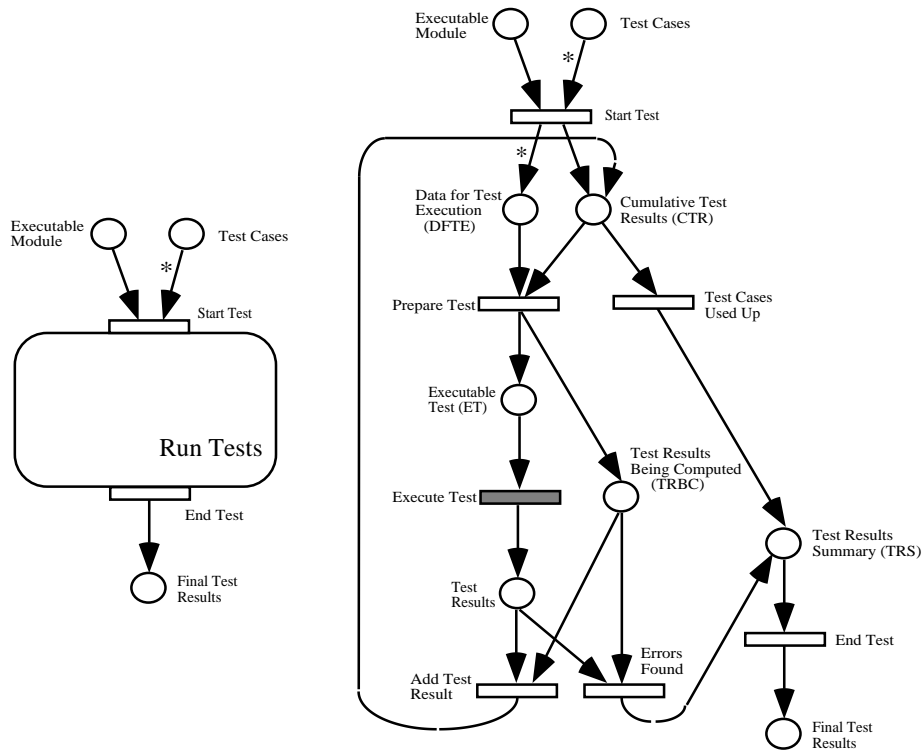


Figure 2: The definition of the interface (left) and implementation (right) of the activity **Run tests**.

An activity state is given by a net marking, i.e. an assignment of tokens to places. Transitions represent events. The occurrence of an event modifies the activity state. The net topology describes precedence relations, conflicts, and parallelism among events. Each activity corresponds to a logical work unit, that may include invocation of other activities and interaction with the environment (tools and humans).

Activity definition. An *activity definition* is an instance of the special type *activity*: it consists of a net where P is the set of places, T is the set of transitions, and A is the set of arcs. An activity has an *interface* and an *implementation* part. The activity interacts with the rest of the process through its interface, which includes a set of starting events, that may initiate the activity, and a set of ending events, that may end the activity. The implementation part, represented by an high-level Petri net, details how the activity is performed by showing the relationships among the events of interest that may occur during its execution. An activity implementation may also contain invocations to other activities. An example of the interface and the implementation of an activity is given in figure 2.

Places and tokens. Each place has a name and a type. A place behaves as a token repository that may only contain tokens of its type or of any of its subtypes. The only way a place may change its contents is by the firing of an input or output transition. A particular kind of place,

called a *user interface* place, may change its content as a consequence of human intervention. User interface places are used to transfer external events caused by humans within the system. The type of a place must be one of the types contained in the *ProcessTypes* set. In particular places can be of type *Activity* (i.e., they may contain tokens whose value is an activity definition), and *Metatype* (i.e., they may contain tokens whose value is a type definition). Consequently, activity and type definitions can be created, manipulated, modified, and deleted as any other values.

Transitions. Transitions represent events whose occurrence takes a negligible amount of time. The occurrence of an event corresponds to the firing of a transition. Each transition has associated with it a guard and an action. The transition's guard is a predicate on input tokens and is used to decide whether an input token tuple enables the transition (an input tuple satisfying a transition guard is called an enabling tuple). The dynamic behavior of a transition is described by the *firing rule*. The firing rule states that when a transition fires, tokens satisfying the guard are removed from input places and the transition's action is executed. As a result of executing the action, an output tuple is inserted in the output places of the fired transition.

A software development process involves the activation of a large variety of software tools. Tool invocation is modeled in SLANG by using *black transitions*. A black transition is a special transition where the action part has been replaced by a call to a non-SLANG executable routine (e.g., a Unix executable file). When the black transition "fires", the routine is executed *asynchronously*. This means that other transitions may be fired while the black transition is still being executed. It is also possible to fire the black transition itself many times with different input tuples, without waiting for each activation to complete. For example, in figure 2 the black transition "Execute Test" is used to represent the execution of an external process running the tests.

Arcs. Arcs are weighted with the number of tokens which flow through the arc at each transition firing. Weights can be statically defined (with a default weight of 1), or dynamically computed. In the latter case, the arc weight is indicated by a "*", and it models events consuming *all tokens* that verify a certain property. In addition of "normal" arcs, SLANG provides two special kinds of arc: *read-only* and *overwrite*. A read-only arc may connect a place to a transition. The transition can read and use token values from the input place in order to evaluate the guard and the action, but tokens cannot be neither removed nor modified. An overwrite arc may connect a transition to a place. When the transition fires, the following atomic sequence of actions occurs: first the output place is emptied of all its tokens, next, the token(s) produced by the firing are inserted in the output place. The overall effect is that the produced tokens *overwrite* any previous content of the output place.

2.2 Mechanisms supporting process evolution in SLANG

In process centered environments, process definitions (or models) play the role of the code in regular software systems. The enactment of the process definition causes the automatic execution of computer-based actions and guides the behavior of people involved in the process.

In SLANG, the process definition code may include not only the model of the software production process, but also the specification of the 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 modeling process evolution.

This section presents the enaction mechanisms of SLANG, with particular emphasis on process evolution.

The *SLANG interpreter* is responsible for the enaction of SLANG specifications. As any other tool, the SLANG interpreter may be called from within the process model, through a black transition. The name *process engine* refers to each running instance of the SLANG interpreter. Whenever the interpreter is called, a new process engine is created. Actually, activity invocation in SLANG may be seen as a derived construct that involves the execution of a call to the SLANG interpreter, via a black transition. Each asynchronous process engine may access shared places during its execution to communicate with other process engines in execution. At the end of execution, the process engine puts the resulting tokens into the places of the invoking activity. Consequently, process engines must be synchronized in order to discipline access to shared and output places in mutual exclusion.

The activity definitions, along with the type definitions used by the activity, are not statically bound to the activity invocation. They are dynamically made available to the process engine at the beginning of the execution of an activity by reading the necessary definitions from places **Activities** and **Types**. This provides the basic interpretive mechanism supporting dynamic evolution.

The mechanisms presented so far provide the ability, within the process model, to manipulate and execute fragments of process definitions. In order to support evolution strategies, it is also necessary to provide mechanisms to manipulate active copies, i.e. the instances of activity definitions that are created during the enactment of the process model.

To manipulate an active copy it is necessary to suspend its execution and make the information about the copy (state, activity and type definitions, etc.) available as a token. SLANG provides a suspension mechanism supporting this functionality: it forces the termination of the process engine, enables manipulation of the active copy, and later spawns a new process engine to restart execution of the modified active copy.

SPADE supports two main classes of change. It is possible to modify definitions, by modifying tokens whose values are activity and type definitions. It is also possible to modify active copies while they are suspended. In most cases, one first modifies a definition and, at some later instant, the effect of a definition change is made visible in a running active copy. In such case, it is useful to distinguish between two times: *change definition time* (CDT) and *change instantiation time* (CIT). CDT is the time at which changes are applied to a SLANG definition; CIT is the time at which a change in a definition becomes visible in a running active copy.

The new definition of an activity can be generated by a meta-process that accesses places **Activities** and **Types**, to edit the required definitions. The time at which editing terminates on a set of definitions defines the CDT of such definitions. The corresponding CITs depend on

the strategy one wishes to adopt: different strategies may be specified in SPADE in order to reflect the effect of a change to a definition in a running active copy; i.e. different strategies are possible to define CIT, given a CDT. A first strategy (*lazy strategy*) does not propagate the modification of activity and type definition to existing active copies. Only when new active copies are created, are the new definitions used. A second strategy (*eager strategy*) is based on the immediate propagation of activity and type definition changes to all existing active copies. Notice that other modification strategies can be envisaged; a whole spectrum of strategies exist from fully lazy to fully eager.

It is also possible to modify a suspended active copy (e.g., its type definitions or its state) without modifying any activity or type definition. This would affect only the changed active copy, with no effect on future creations of active copies of the same or other activities.

3 SPADE data description and classification

This section provides the description and classification of data that will be stored in the SPADE repository. All the information describing a SLANG specification (*ProcessTypes* and *Process-Activity*) and the process state (all instances of *ProcessData* and its subtypes) are stored in a repository that is shared by all the process engines. The repository is built on top of O_2 . The SLANG interpreter uses the database to access both the description of the process model and the process data produced and modified as result of its enactment.

The SPADE repository is organized according to the following structure.

- The schema of the database is partitioned in two parts: a *fixed part* that contains the types of SLANG basic constructs (including type *Activity* and type *Metatype*), and a *modifiable part* containing the definition of the types used within a specific process model (all subtypes of *ProcessData* type). This modifiable part can change to cope with modification in the modeled process. For example, we may add a type to describe a new class of documents or software items.
- At the instance level, we have two different sets of objects as well. The instances of the types in the fixed part of the schema correspond to a specific process model definition (i.e., a collection of arcs, transitions, and places constituting a SLANG specification). The instances of the modifiable part of the schema correspond to process data (e.g., modules, test results, test cases, etc.) produced or modified during process enactment.

Summing up, we have the scenario described in Figure 3. It is not possible to change the definition of the SLANG language (fixed part of the schema). Changes to the variable part of the schema and to the instances of the fixed part correspond to changes in the process model. Changes to the instances of the modifiable part correspond to changes in the state of the enacted process model.

4 Database Requirements

A first set of SLANG features that poses specific requirements to the OODB is the following:

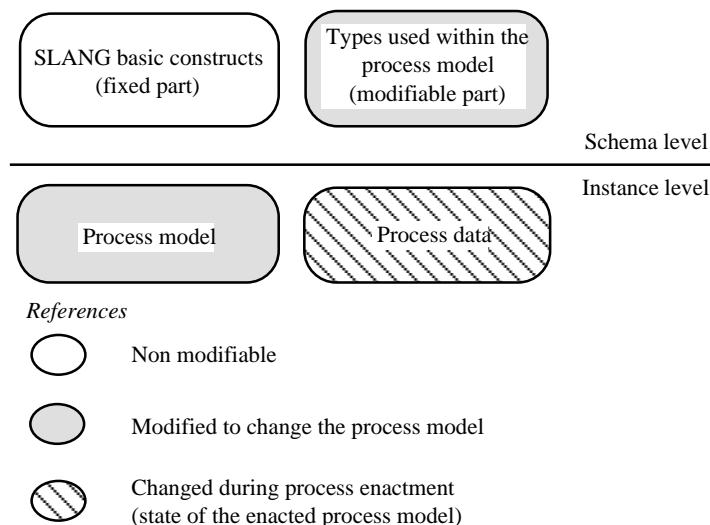


Figure 3: Structure of the SPADE repository.

1. The process engine is a program that depends only on the types in the fixed part of the schema. It must not depend on the types of the modifiable part, because they may be changed both during the process definition and during process enactment (process evolution): it is clearly not feasible to regenerate the process engine whenever the process manager modifies the running model.
2. The database schema may be changed at run-time. This means that it must be possible to apply changes to the schema concurrently with the execution of models that are instances of the same schema. Suitable mechanisms have to be established in order to ensure the proper degree of synchronization and consistency.
3. Since the definition of types in the modifiable part may change, the DBMS has to support migration of the existing objects from the old definition of their type to the new one.
4. The interpretation of SLANG specific constructs must be effectively supported.

The rest of this sections describes the database requirements deriving from the specific features of the process language described above.

The following discussion is based on the class hierarchy described in figure 4. The graphical convention used is the following: boxes represent classes, where the upper part contains the name, and the bottom part contains relevant properties; arrows represent inheritance relationships.

The given hierarchy is a refinement of the general hierarchy depicted in figure 1. It is a simplified, although likely, representation of the SLANG class hierarchy: Place and Token are classes belonging to the fixed part of the schema (i.e. to the language definition), while IntToken, StrToken, IntPlace and StrPlace are sample specializations of language elements (i.e. classes used to define the specific process).

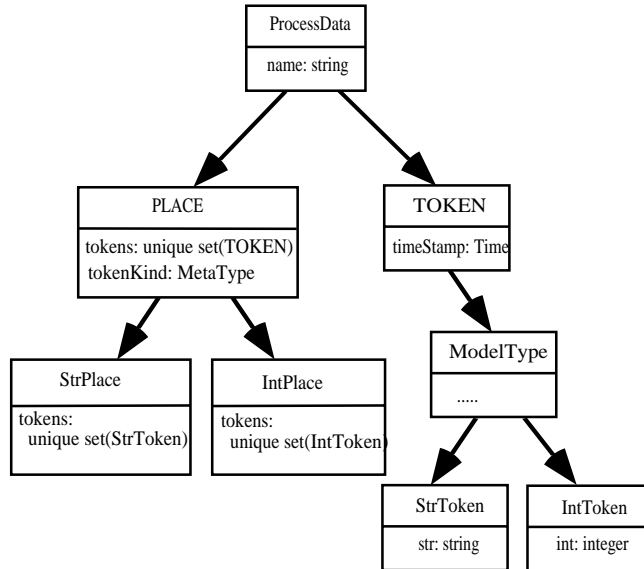


Figure 4: Language constructs class hierarchy.

4.1 Dynamic interpretation of queries

The process engine executes the SLANG net specification by computing at each execution cycle the set of transitions that are enabled (i.e., whose guards evaluate to true) and selecting a transition to fire. The firing of a transition involves the execution of the associated action. The guard and action specifications are loaded at run-time from the OODB and then interpreted. Each process engine must be able to execute queries and update operations on the database. The code to be executed may be changed by the user concurrently with the execution of the model: once a change has been committed, the process engine will use the new definition, when the modified transition has to be fired. In other words, each time the interpreter retrieves a piece of code from the DB (this is done e.g. at each activity initiation) such code may have changed with respect to the previous execution. The process engine must be able to use the last version of such code stored in the database.

There are several kinds of changes in the overall model that may be performed during process execution, in particular they may regard:

- the types in the modifiable part;
- the structure of the model (i.e. the topology of the net);
- the text of a guard (or that of an action).

Every time a transition is fired, SLANG executes the action associated with that transition. The effect of the action is the creation of tokens in the output places and the assignment of values to these tokens' attributes. Such values are either taken directly from the input places or computed by the transition action. We remind that tokens represent the artifacts of the software process (e.g. code, specification documents, error reports, etc.) and process specific data

(e.g. the time elapsed from the beginning of the project, amount of resources expended, planned completion date, etc.). The description of the artifacts (i.e. their type) can be determined only when the process is instantiated (i.e. it is independent of the language) and may be changed during the execution of the process (for example to accomplish the decision of enhancing the information contained in a report).

Because of these features, the implementation of SLANG actions requires the ability to build objects of a class that is not known at compile time, i.e. we want to be able to define a new class and to create objects of that class without stopping and recompiling the process engine.

The most natural way of accomplishing the execution behavior described above is to interpret the model. Interpretation is easy if the OODB provides a programmatic interface similar to what it is offered by several relational databases, i.e. if it is possible to invoke the OODB query and manipulation language interpreter through a procedure call, passing as a parameter the string of text representing the required operation. In SQL-based systems, for example, this feature is called *dynamic embedded SQL*.

Using this kind of facility we should be able to build an interpreter of dynamically changing code, since we can define at run-time queries implementing transition guards and actions, independently of the process model we are enacting.

Since transition actions may involve the creation of new objects, the interpreter and the language must also be able to accept object creation statements. In fact, the language should be computationally complete, in order to represent transition actions.

4.2 Schema management

The definitions of types that belong to the modifiable part of the schema (i.e. all types but those defining the elements of the language) may be changed. It must be possible to change these definitions even during process enactment, i.e. when they are used by some process engine.

Moreover, schema updates must be safe: each change must leave the database in a consistent state, avoiding to loose information and, consequently, to cause run-time inconsistency and abortion of transitions. In particular, when a type definition is changed, we must handle the instances of the changed type: we need some mechanism to support migration of the existing objects from the old definition to the new one.

Although it is possible to explicitly convert existing objects, this operation is rather cumbersome, and in general this solution is unfeasible for databases of non trivial size. An automatic mechanism for type migration should support different operating requirements:

- lazy migration: the object is “converted” to the new type only when the object is accessed;
- eager migration: the object is “converted” to the new type as soon as the new type is defined.
- any user-defined migration being an intermediate strategy between fully eager and fully lazy migration.

A complementary approach to type updating consists of providing support for type versioning: both types survive, new objects will be created according to the new type definition, while old

objects are accessed according to the corresponding type version.

4.3 Distribution

SPADE supports multi-user project development by allowing the distribution of running activities over a number of workstations. The model of the process that supervises and governs the activities is shared among the workstations.

There are, basically, two ways to implement this situation. The first consists of allowing for a distributed access from the users' workstations to a DB server. The whole model resides on a server, that is a fundamental component in order to determine the performance of the whole system. This approach appears to be interesting for quite small projects, and for large projects that can be split into quite independent sub-projects.

The second idea, suitable for larger projects, is to distribute transparently the process model execution over various workstations. This implies distributing the process data, consisting mainly in the artifacts that are produced locally by each user. The tools used should not need to know anything about the physical distribution of the data used: it should be the responsibility of the database system to manage physical distribution. This approach reduces the client-server traffic on the local area network, while it introduces the need for consistency control mechanisms to preserve the consistency of the distributed databases.

4.4 Other issues

In this section we briefly describe some features that are frequently mentioned as requirements for a software engineering database, but play a minor part in our environment, at least at the present stage of development. In particular, we describe why at the moment such requirements have not been considered in the first implementation of SPADE.

4.4.1 Concurrency management

Basic transactional mechanisms (such as two-phase lock) must be available, in order to achieve atomicity of actions. They are needed to ensure that the different process engines access the database in a controlled and synchronized way.

More sophisticated concurrency control mechanisms, such as long or nested transactions, are not needed, since the behavior they provide can be achieved by coding the transaction behavior in the net.

4.4.2 Access control

Each user has an identity and a role: these attributes can be modelled explicitly in SLANG, and guards may take them into account, so that only users having a specific profile are enabled to perform given actions. In other words, access control is explicitly programmed by the process modeler.

The database management system has simply to provide basic mechanism that prevent unauthorized access to the object base, e.g. from outside the process centered environment.

4.4.3 Versioning

Versioning of artifacts can also be “programmed” in SLANG, thus a specific support by the database is not strictly required. Support from the database could facilitate the process programmer in writing the definition of the versionable artifacts.

5 Using O_2 as a process repository

In this section we briefly describe O_2 , a “state of the art” object-oriented database management system, and we assess its suitability as the repository supporting a process centered development environment.

The O_2 OODBMS

The first prototype of O_2 was the result of a research project started in 1986 by the Altair consortium; O_2 is now a commercial product of O_2 Technology.

O_2 is provided with a complete development environment and a set of user interface tools. Information is organized in *objects* (instances of *classes*), and *values* (instances of *types*). A value has only a type, while an object has an identity, a value and a behavior, determined by the *methods* defined in its class. Methods are coded in O_2C , a fourth generation language, born as a superset of ANSI C, extended to support the object-oriented data model of O_2 .

O_2 allows the user to write programs, to manipulate his/her data base and to generate an appropriate user interface. However applications written in other languages, like C and C++, have access to the features offered by O_2 by means of an import/export mechanism. The user can also easily design graphic interfaces using O_2Look , built on top of *X Window System* and *Motif*, that provides a set of high-level functions to display and to edit complex objects.

The O_2 system [14, 8] offers a declarative query language, called O_2SQL , whose syntax is styled on SQL, the standard query language for relational database. O_2SQL allows the user to query an O_2 database either in an interactive way or under program control.

5.1 The process engine and the dynamic interpretation of SLANG

The basic part of the SLANG interpreter (i.e. the guard evaluation - transition selection - firing loop) has a rather traditional structure, and is therefore written in O_2C .

The dynamic interpretation of guards and actions described in section 4.1 is achieved by providing the class Transition with two methods, evaluateGuard and executeAction, that call the O_2SQL interpreter for the evaluation of a query whose text (defined at run-time) represents the guard or the action. The O_2C code of the process engine that calls the O_2SQL interpreter is reported in figure 5.

It is quite obvious that the aforementioned methods must be general, i.e. they must be able to deal with any guard or any action.

Therefore, a first requirement is that the call of the query evaluator must not depend on how many places are in input to the transition whose guard we are evaluating. This need is taken into account by declaring the **parameters** of the query as a list of Places, i.e. the `o2query` call

```

o2 list(unique set(unique set(unique set(Token)))) result;
o2 list(Place) parameters;
...
o2query(result, query_code, parameters);

```

Figure 5: Call of the O_2SQL interpreter.

will receive any number of instances of any subtype of Place.

It is also required that the result of the query is a general structure (i.e. it is valid for any number of input places, and for any weight of input arcs). The definition of the `result` given in figure 5 meets this requirement. Its meaning is related to the concept of “enabling tuple”, i.e. the set of sets of tokens that satisfies the guard predicate. Each set, whose cardinality depends on the weight of the connecting arc, is taken from a different input place. The inner unique set of `result` represents the set of tokens received from each input place. The set of such sets (i.e. the enabling tuple) is represented by the intermediate unique set. The outer unique set contains all of the enabling tuples.

Although in principle the solution presented in figure 5 solves the problem of dynamic interpretation of the guards, there are a couple of problems in the implementation of O_2SQL that forced us to modify it.

O_2SQL performs a precompilation of the code to be interpreted, in order to optimize the evaluation of the query. In this step, any parameter of type collection (i.e. sets, lists and unique sets) is considered as a collection of objects that are bound to the static type of the collection. For example, in figure 5 `parameters` is considered as a non-polymorphic list of Places: if the `query_code` refers to any feature of a subtype of Place (as is often the case, since the `query_code` is built knowing the actual type of the Place) it is rejected by the precompiler. In order to overcome this problem, method `evaluateGuard` has to contain an `o2query` call with as many parameters (each one representing an input place) as the maximum fan-in of any transition in the net. Figure 6 reports a piece of code of method `evaluateGuard`. The example refers to the evaluation of the guard of a transition having two input places, the first one containing objects of type `IntToken` (see figure 4). Note that the code of the guards explicitly mentions attribute `int`, that is exclusive of type `IntToken`.

Since `parameters` (`parameters[0]`, `parameters[1]`, etc.) are objects, there are no typing problems.

O_2SQL does not support the creation of new objects, as needed by the execution of an action. In order to solve this problem, we have devised the following solution:

- We have explicitly built the Meta Schema of the SLANG interpreter, defining the class *MetaType* that has particular properties in order to handle token types. Each instance of class *MetaType* has an attribute `prototype` that stores a template instance of the particular token type described by the *MetaType* instance itself.
- When a place is created the attribute `tokenKind` has to contain a reference to the *MetaType* instance that describes the type of tokens that the place has to store. For example, when a new place of class *Specifications* is created, the attribute `tokenKind` of that place is

```

o2 list(Place) parameters;           /* the list of input places */
o2 unique set(list(unique set(Token))) result; /* enabling tuples */

/* let self->Guard be
   "select distinct list(unique set(t11, t12), unique set(t21))\
   from t11 in $1.tokens,\
       t12 in ($1.tokens - unique set(t11)),\
       t21 in $2.tokens\
   where forall t1 in unique set(t11, t12): (t1.name = t21.name and t1.int = 3)"
*/
o2query(result, self->Guard,
        parameters[0], parameters[1], parameters[2], ....);

```

Figure 6: Passing many parameters to the O_2SQL interpreter.

associated with the *MetaType* instance that defines the class *SpecificationDocument*.

- In class *Token* we define two methods, `duplicate` and `makeAssignments`, that will be used to create copies of the prototype token, and to assign its attributes. These methods are redefined in each subclass of *Token*.

For example, if we have to create a new *SpecificationDocument* in an output place, we make a copy of the *SpecificationDocument* prototype attached to the object related to the attribute `tokenKind` of the place. This is done by means of the correct `duplicate` method code. The resulting *SpecificationDocument* attributes are then assigned the desired values by means of the (dynamically bound) method `makeAssignments`.

5.2 Run-Time schema modification

The version of O_2 that we used to make the experiments reported in this paper (release 4.0) offers the possibility to manipulate the schema only by means of the DB's alphanumeric interface. It is not possible to directly manipulate the schema using O_2 's programmatic interface, i.e. it is not possible to invoke the database schema manipulation capabilities at run-time. The only possible way to add new classes to the DB schema is to write the appropriate DDL (Data Definition Language) code and compile it as an independent process. However, the application does not need to be interrupted, because a process could run the application, while another process updates the schema.

Obviously, an extension of the O_2 supporting schema manipulation would greatly facilitate the accomplishment of this task. In order to cope with this requirement the new version of O_2 [15] allows the direct vision and manipulation of the predefined object *Meta Schema*, describing the schema of the used base. This feature facilitates the run-time creation or modification of classes, solving the problem shown above. Furthermore these functions should intercept possible schema inconsistencies, letting the programmer avoid this kind of run-time errors. Another interesting usage of the Meta Schema handling is the ability to build and to use objects of a class that is

not known at compile time, as is required by the definition of SLANG actions.

5.3 Type migration

O_2 does not provide any support to type migration. Any change made to the type structure of a class (adding or removal of an attribute, for example) will yield inconsistent objects of that class in each base governed by the schema. This inconsistency will lead to wrong accesses during the execution of a body and abortion of transactions. In order to avoid this problem, the user may *dump* and successively delete all objects of this class before changing its structure. Dumped information can be later retrieved and associated to instances of the updated type.

The direct handling of an application meta-schema, offered by the new version of O_2 , enables the programmer to write a function that gets an object a of class A , creates B , a new subclass of A , and an instance b of B , derived from a . B could be a new version of A : in this case the function may initialize the new objects with the corresponding values found in the source objects and the remaining attributes with default values.

This solution is probably not very fast (since all the existing references to a have to be searched for and transformed into references to b), but it solves the problem satisfactorily, since type updates are supposed to be rather unfrequent.

An effective version manager would obviously be preferable.

5.4 Distribution

O_2 is based on a client-server architecture: this means that clients (that are responsible for the execution of methods and applications) may run on different machines, while the server (that is responsible for object management) runs on a centralized machines. Although not an ideal situation, this architecture is sufficient to support a distributed SDE with a limited number of people (clients) involved in the process. The number of clients is limited only by the amount of traffic required to move objects to and from the server.

Logical distribution of data (i.e. separate bases having a common schema) will be supported by O_2 in a near future.

6 Concluding remarks

In this paper we have reported our experience in building the repository for a process centered software development environment using O_2 . We illustrated the requirements that a PSDE poses to the supporting repository, and we described how O_2 fulfils such requirements.

Our experiment gave encouraging results, that let us undertake the development of an interpreter for SLANG using O_2 . However, several goals are still to be met, as discussed in section 5. Among these, we remind the following:

- fully functional embedded query language;
- support for type migration;

- physical distribution of data.

We hope that this paper will contribute to enhance the confidence of the software engineering community in the capabilities of object-oriented databases, as well as to provide some suggestions for the evolution of the object-oriented database technology.

Future activities include:

- Experiments aiming at determining when physical distribution of data is actually necessary. This will imply evaluating the limits of the client-server architecture of O_2 (i.e. when it is actually necessary to store data where it is most frequently needed) and investigating the efficiency issues related with page caching at the clients.
- Integration of tools in the SPADE environment, investigating both the usage of data type-aware tools, and the current development environment tools. It would also be interesting to integrate the process engine and a tool environment such as Field [18], in order to couple the benefits of a PSDE and of a service oriented, message driven tool environment.
- We will also continue to observe the evolution of OODBMS, in order to identify possible features that would allow the construction of more powerful or more efficient repositories for SDEs. It is our will to experiment with such new features, and possibly to use them in the development of the repository for SPADE.

Acknowledgements

People from O_2 Technology provided continuous support and invaluable help.

References

- [1] M. Atkinson, F. Bancilhon, et al. The object-oriented database system manifesto. In *Proceedings of the First DOOD Conference*, Japan, 1989.
- [2] Sergio Bandinelli, Luciano Baresi, Alfonso Fuggetta, and Luigi Lavazza. Requirements and Early Experiences in the Implementation of the SPADE Repository. In *Proceedings of the 8th International Software Process Workshop*, Berlin (Germany), February 1993.
- [3] Sergio Bandinelli and Alfonso Fuggetta. Computational Reflection in Software Process Modeling: the SLANG Approach. In *Proceedings of the 15th. International Conference on Software Engineering*, Baltimore, Maryland (USA), May 1993.
- [4] Sergio Bandinelli, Alfonso Fuggetta, and Carlo Ghezzi. Software Processes as Real-time Systems: a Case Study Using High-level Petri nets. In Alfonso Fuggetta, Reidar Conradi, and Vincenzo Ambriola, editors, *Proceedings of the First European Workshop on Software Process Modeling*, pages 203–226, Milano (Italy), May 1991. AICA - Italian National Association for Computer Science.

- [5] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Sandro Grigolli. Process Enactment in SPADE. In Jean-Claude Derniame, editor, *Proceedings of the Second European Workshop on Software Process Technology*, volume 635 of *LNCS*, pages 67–83, Trondheim (Norway), September 1992. Springer-Verlag.
- [6] Sergio Bandinelli, Alfonso Fuggetta, and Sandro Grigolli. Process Modeling-in-the-large with SLANG. In *Proceedings of the 2nd International Conference on the Software Process*, Berlin (Germany), February 1993. IEEE.
- [7] P.A. Bernstein. Database system support for software engineering - an extended abstract. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 166–178. IEEE, 1987.
- [8] O. Deux. The O₂ System. *Communications of the ACM*, 34(10), October 1991.
- [9] S. Dewal, W. Emmerich, and K. Lichtinghagen. A Decision Support Method for the Selection of OMSs. In *Proceedings of the Second Int. Conference on System Integration*, pages 32–40, Morristown, N.J., 1992. IEEE Computer Society Press.
- [10] S. Dissmann, W. Emmerich, B. Holtkamp, K. Lichtinghagen, and L. Shope. OMSs comparative study. Internal Report D2.4.3-rep-1.0-UDO-EL, ATMOSPHERE, 1991.
- [11] Wolfgang Emmerich, Wilhelm Schäfer, and Jim Welsh. Suitable Databases For Process-centred Environments Do Not Yet Exist. In Jean-Claude Derniame, editor, *Proceedings of the Second European Workshop on Software Process Technology*, volume 635 of *LNCS*, pages 94–98, Trondheim (Norway), September 1992. Springer-Verlag.
- [12] Carlo Ghezzi, Dino Mandrioli, Sandro Morasca, and Mauro Pezzé. A Unified High-level Petri Net Formalism for Time-critical Systems. *IEEE Transactions on Software Engineering*, February 1991.
- [13] Watts S. Humphrey. *Managing the Software Process*. SEI Series in Software Engineering. Addison-Wesley, 1989.
- [14] C. Lecluse, P. Richard, and F. Velez. O2, an object-oriented data model. In *Proceedings of SIGMOD '89 - Int. Conf. on the Management of Data*, pages 424–433, Portland, OR, 1989. ACM.
- [15] O2 Technology, 7 rue du Parc de Clagny - 78035 Versailles Cedex, France. *The O2 User's Manual*, January 1993. Version 4.2.1 - Chapter 11.
- [16] Leon Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*. IEEE, 1987.
- [17] M. H. Penedo and C. Shu. Acquiring experiences with the modelling and implementation of the project life-cycle process: the PMDB work. *Software Engineering Journal*, pages 259–273, September 1991.

- [18] S. Reiss. Connecting Tools using Message Passing in the FIELD Program Development Environment. *IEEE Software*, pages 57–67, July 1990.
- [19] The GoodStep team. Description of software engineering applications and requirements for an object-oriented repository. Deliverable 1, ESPRIT project 6115 GoodStep - General Object-Oriented Databases for Software Processes, March 1993.