A Fine-grained Process Modelling Experiment at British Airways[*]

Jim Arlow
British Airways Plc
TBE (E124)
Viscount Way
Hounslow, UK
Jim.Arlow@btinternet.com

Sergio Bandinelli
ESI
Parque Tecnológico, 204
48170 Bilbao
Bizkaia, Spain
Sergio.Bandinelli@esi.es

Wolfgang Emmerich
City University
Computer Science
Northampton Square
London EC1V 0HB, UK
we@city.ac.uk

Luigi Lavazza
Politecnico di Milano
and CEFRIEL
Via Emanueli 15
20126 Milano, Italy
lavazza@mailer.cefriel.it

Abstract

We report on the experimental application of process technology that we did at British Airways (BA) as part of the GOODSTEP project. The goal of GOODSTEP was to enhance and improve the functionality of an object database management system (ODBMS) to yield a platform suited to the construction of process-centred software engineering environments (PSEEs). These enhancements were exploited and validated by the construction of the GOODSTEP framework for PSEE construction, which includes the SPADE software process toolset. We used the process modeling language SLANG to model BA's C++ class library management process, and we constructed an experimental PSEE based on SPADE. BA required processes to be automated at a finer degree of granularity than that of tool invocation. We have demonstrated that SLANG and SPADE offer the basic mechanisms for modelling these fine-grained processes. We have also shown that it is feasible to generate tools for dedicated processes and integrate them within a SLANG model so as to facilitate fine-grained process automation. However, our experience highlighted some open problems. For instance, SLANG process models are tuned to efficient enactment, thus containing very detailed process fragments. These are not the most appropriate representations for humans trying to understand the process model. Although the airline did not deploy the PSEE in its production environment, the experiment proved beneficial for BA because the modelling activity itself uncovered serious flaws in the existing process.

Keywords

Software Process Modelling Experiment, Process-centred Software Engineering Environment

1 Introduction

During the past decade, a great deal of research has been devoted to process technology. Process modelling languages have been defined in order to specify software processes on a formal

[*]This work has been partly funded by the CEC within ESPRIT-III project 6115 (GOODSTEP). The work was done while W. Emmerich was with University of Dortmund (Germany), S. Bandinelli was with CEFRIEL (Italy).

basis. Examples of these languages are extensions to programming languages (e.g. extensions of Ada [Sutton et al., 1990] and Prolog [Peuschel et al., 1992]), Petri net based approaches (FUNSOFT [Emmerich and Gruhn, 1991] and SLANG [Bandinelli et al., 1993b]) and multi-paradigm approaches integrating several high-level descriptions (ESCAPE [Junkermann, 1995] and SOCCA [Engels and Groenewegen, 1994]). Process modelling environments have been constructed for these languages so as to edit, simulate, analyse, enact and evolve process models. Examples include Merlin [Peuschel and Schäfer, 1992], SPADE [Bandinelli et al., 1993a], Melmac [Deiters and Gruhn, 1990] and Marvel [Barghouti and Kaiser, 1990]. A central component of these environments is a process engine that interprets a process model. A number of attempts have been made to build environments that, in addition to process modelling components, include tools for the actual software development. These tools are integrated with the process engine so as to provide services for process automation and to inform the engine about process-relevant events that they have captured. Such environments are known as *process-centred software engineering environments* (PSEEs).

While development of process technology has attracted a vast amount of effort, only a small amount of attention has been paid so far to the industrial application of the facilities developed. A notable exception is [Dinkhoff et al., 1994] where FUNSOFT nets have been applied to large-scale business processes in the area of real estate management. The nature of these business processes, however, is considerably simpler than those of software processes.

The contribution of this paper is an account of the experience that we gained when we applied the SLANG process modelling formalism and the SPADE environment to the modelling and enactment of a software process in an industrial setting, namely at British Airways (BA). BA is a large software developer in the UK, with some 2,000 IT staff. To increase productivity and quality, BA have founded a group called *Infrastructure* who is in charge of maintaining the design, implementation and documentation of reusable C++ class libraries. SLANG was used to capture, model and improve the class library development and maintenance process.

The process was not only modelled, but also supported with a customised PSEE, the British Airways SEE. This PSEE integrates the SPADE process engine with tools for the development of Booch class diagrams, C++ class interface definitions, C++ class implementations and class documentations. These tools were generated with the GENESIS tool construction toolset [Emmerich et al., 1997a]. The integration of tools and process model was done in a way that facilitates process guidance at a finer level of granularity than tool invocation.

SPADE and GENESIS were developed in the GOODSTEP project [GOODSTEP Team, 1994]. The experiment reported here was also carried out within that project so as to validate the GOODSTEP tools. Section 2 briefly describes the GOODSTEP project. Section 3 outlines the goals of our process modelling experiment. It is followed by a discussion of the baseline of the experiment, i.e. the existing process at British Airways, the BA SEE tools that have been generated and the SLANG process modelling language. Section 5 presents the way the process modelling experiment was conducted and the results specific to the problems of British Airways. Section 6 discusses the implications of the lessons we learned for process modelling and process-centred environments in general. Finally, we indicate in Section 7 open research problems highlighted by our experience and indicate how our current work contributes to the problems identified.

2 The GOODSTEP Project

GOODSTEP [GOODSTEP Team, 1994] brought together European expertise in the areas of databases and software engineering. GOODSTEP started in September 1992 and was successfully completed in November 1995. The project delivered an advanced database management system and a development framework for the construction and customisation of PSEEs.

The baseline of the project was an existing European object-oriented database product: O_2 [Bancilhon et al., 1992]. Rather than developing a new system from scratch, GOODSTEP enhanced and improved O_2. The choice of an object-oriented database management system as a starting point for the project derives from the inadequacy of relational database systems for engineering applications, which has been recognised for some time [Maier, 1989].

We have shown in [Emmerich et al., 1993a] that the O_2 product was well adapted to meeting the requirements [Emmerich et al., 1993b] for storing process and product data of a PSEE. O_2's schema definition language supported the fine-grained definition of documents and relationships among them; O_2 provided a query language and a meta schema that are necessary primitives for the implementation of process reflexivity [Bandinelli et al., 1993a]; O_2 also supported the transactions that are required for the preservation of integrity of documents and process information in the face of hardware or software failure and for low-level concurrency control. Finally, O_2's client/server architecture provided limited support for distribution.

O_2, however, did not fulfill all requirements for the construction of PSEEs. In order to accomplish the construction of SPADE on top of O_2, it had to be extended with facilities for schema updates, primitives for version management, and concurrency control based on objects rather than disk pages, that were used as secondary storage unit.

The implementation of reflexive capabilities in a process modelling language with a static type system requires the ability to create new types in the database schema as well as to change existing types during the enactment of the process, possibly migrating the existing instances to the new type definitions, or compiling at run-time newly created or changed methods. Schema update facilities that address these requirements [Ferrandina et al., 1994, Ferrandina et al., 1995] were introduced in O_2 by GOODSTEP and are now part of the O_2 product.

The early phases of GOODSTEP have also extended the O_2 ODBMS with basic primitives for version management of composite objects [Delobel and Madec, 1993]. This feature, which is also available in the current O_2 product version, makes process modelling easier and more effective.

Originally, not only client/server communication, but also concurrency control, which implements ACID transactions, were page-based since the server was not aware of the objects that resided on the pages it was managing. Situations occurred with small objects where the page-level concurrency control revealed conflicts even though the concurrent transactions were accessing disjoint sets of objects, just because they resided by chance on the same page. Conflicts caused transaction abortions and even deadlocks, causing loss of efficiency and requiring specific code to manage spurious deadlocks. To avoid such conflicts, O_2 was extended with object-level concurrency control. Concurrency control is, by default, still based on page locks. As soon as a conflict occurs, however, the concurrency control switches to object-level locking. The features of the extended version of O_2 delivered by GOODSTEP were extensively exploited in the construction of SPADE, as described in [Bandinelli et al., 1995a].

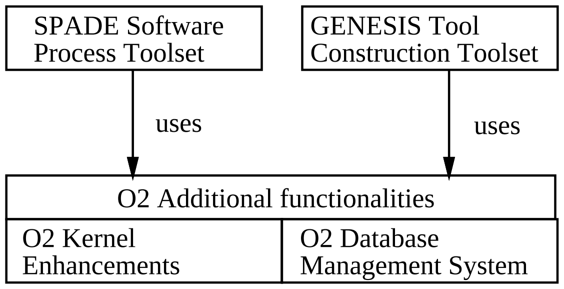

SPADE Software Process Toolset
GENESIS Tool Construction Toolset
uses
uses
O2 Additional functionalities
O2 Kernel Enhancements
O2 Database Management System

Figure 1: Components developed in GOODSTEP

These ODBMS extensions were then exploited for the development of a framework for the construction of PSEEs of which a coarse-grained overview is displayed in Figure 1. That framework consists of two components, the SPADE software process toolset with the process modelling language SLANG and the GENESIS tool construction toolset. GENESIS includes a compiler for the GOODSTEP Tool Specification Language (GTSL). SPADE exploits the extended ODBMS for storing process models and process states. GENESIS generates ODBMS schemas and applications that are used to create and modify software documents.

3 Goals of the Experiment

The goals for this experiment were manifold and can be considered from the points of view of technology providers and technology users.

The motivation of the technology providers in this experiment was to evaluate process technology in an industrial setting. The goals of the experiment are reflected in the following questions:

Feasibility: Is it feasible with the language primitives available in SLANG and GTSL (the GOODSTEP Tool Specification Language) to lower the level of granularity of process models, in order to improve process automation?

Scalability: Is SLANG sufficiently scalable to handle complex processes such as those that occur in industrial practice? Are the resulting process models of any use for supporting communication among the developers involved?

User Acceptance: Does the resulting process model provide developers with sufficient freedom to express their creativity, or does it impose strict constraints that could make the development harder and less productive?

Performance: Is the enactment of a fine-grained process fast enough, or does it slow down users more than it helps?

The main motivation of technology users was to understand what process technology can do for them. They were interested in:

Process Understanding: The Infrastructure Group wanted to check whether, after having worked for two years on the maintenance of class libraries, it had reached a common (i.e. consistent) understanding of the process.

Process Improvement: The group knew the flaws in their process and wanted to remove them. They were curious to see how formal process modelling could help.

Process Automation: Several tasks of the group were done manually and the group was fascinated by the idea that an environment customised to their particular needs might automate a relevant part of the work.

4 Baseline of the Experiment

In the first part of this section we discuss the problem space, i.e. the library development process to be modelled and automated. We then discuss the technology available for solving the problem. In the second subsection, we discuss the GENESIS tool generation facility that has been developed in GOODSTEP and in the third subsection we discuss the facilities available in SPADE for process modelling and enactment.

4.1 Existing Library Development Process at BA

The Infrastructure Group has created a process handbook entitled *Standard Development and Release Procedures.* It elaborates on the process to be used for class library development and maintenance. The handbook suggests, in a rather informal manner, the various actions to be taken for class library development and maintenance. It identifies a number of document types, including Booch class diagrams, C++ class interface definitions, C++ class implementations, class documentation, Makefiles, configurations for dynamic linkage and the like.

Different developers in Infrastructure have different roles. Some developers are *programmers* responsible for implementing and documenting classes in particular libraries. To date, one developer is a *QA engineer*, who is in charge of approving new or changed libraries. A *librarian* administers the different versions contained in library configurations.

Infrastructure identified two main problems with their approach to process management. The first problem is that their informal definition of the process easily leads to misunderstandings. Two Infrastructure developers, who have worked in the same office for two years, discovered a serious misunderstanding of a key concept defined in the handbook when we discussed their process in a meeting. It turned out that they did not have a shared understanding of the semantics of a library configuration in *beta test* as opposed to *development* mode. The second problem is mentioned in the following quote from Infrastructure members:

> *"Within Infrastructure, we have had to apply an excessive amount of effort to establish change control procedures, but these procedures are only partially effective as they are not enforceable by our current toolset. As BA's stock of reusable components grows, the problems of effective change control will become more and more pronounced." [Arlow et al., 1994]*

An obvious approach to enforcing change control procedures is to model them with a process modelling language and to guide the process by subsequently enacting the model constructed. This requires development tools to be integrated into the process environment in such a way that tools cannot be abused in order to circumvent the modelled process. Moreover, the integration of tools and process has to be fine-grained, at least partially, so that the process model can react to the execution of individual tool commands, rather than complete tool sessions. The reason for this is that document contents defined in earlier stages determine documents to be produced in later stages. In the BA process, a class icon included in a Booch design requires creation of documents for a C++ class interface definition, a C++ method implementation and class documentation. As documents are generally considered as first class process modelling concepts, the process model should at least be notified about document creation, if not be responsible for the creation itself. Therefore, a tool command for the creation of a component of one document, for which an inter-document consistency constraint requires the creation of another document, might have to be integrated with the process model.

We modelled the BA class library development and maintenance process with SLANG. Then a dedicated set of tools tailored towards the particular needs of Infrastructure was generated. This toolset was integrated with the SLANG process model to enforce the process and to support process automation at the required levels of granularity.

4.2 Tool Specification and Generation using GENESIS

The need for rapid tool construction through a tool generator is motivated by the observation given above that enforcement of process definitions, such as change control procedures, require *process sensitive* tools to be used. Process sensitive tools interact with a process model through a joint communication protocol, and thus contribute to the implementation of a process model.

A flexible tool construction mechanism is required to develop tools for use with particular process models. GTSL was defined as a high-level language to accomplish rapid tool customisation and construction. Tools for the BA SEE have been generated from GTSL specifications [Emmerich et al., 1997a].

4.2.1 Specification of Tools in GTSL

GTSL specifications of environments use the object-oriented paradigm to define the tools and documents. The use of object-oriented concepts is motivated by the fact that tool specifications can become considerably complex and need to be properly structured to keep such complexity manageable. Moreover, a number of properties re-occur in different parts of tool specifications. Using object-oriented principles, these properties can be specified in abstract specification components and are then inherited by more concrete components.

Due to the heterogeneity of the different static and behavioural concerns of tools, it is impossible to find a unique formalism that expresses these concerns at an appropriate level of abstraction. Instead, we separate the different concerns and offer the most appropriate formalism for each of them. We integrate these different formalisms into a domain-specific *multi-paradigm language* that uses rule-based, object-oriented and imperative concepts.

A GTSL environment specification is structured into a number of *tool configurations*. Each of these configurations consists of a number of *classes* that define the different node types occurring in project-wide abstract syntax graphs [Emmerich et al., 1993b]. Different *sections* are provided to define properties of a class such as attributes, abstract syntax and semantic relationships. Static semantics and inter-document consistency constraints of documents are specified in a *semantic rule section*. The available operations to modify graph nodes are defined in a *method section*. The invocation of operations from commands and the availability of commands are specified as patterns in *interaction sections*. *Multiple inheritance* enables properties from superclasses to be reused in subclasses.

4.2.2 Integration Facilities in GTSL

Tools generated from GTSL specifications can not only be used through a user interface. They also offer services to their environment. These services can be used for tool integration purposes and, in particular, for the integration of tools with a process model. We distinguish between *generic* and *tool-specific services*. GTSL defines about 20 generic services, examples of which are the creation of a document of a particular type, opening a particular version of a document or the computation of a printable representation of a document. Generic services are supported by any tool generated from a GTSL specification and are implemented by the GTSL run-time system. Tool-specific services are specified by the tool builder. They are declared in tool configurations [Emmerich, 1996b] and specified in the same style as tool commands.

Events allow the tool to inform the environment about certain incidents. An event can either be a *request* or a *notification*. A request is sent to the process engine, which either grants the request or rejects it. Requests are used in interactions of BA SEE tools to ask the process engine for the permission necessary to perform certain activities. A notification informs the receiver about a certain action, but does not await its response.

As examples, consider the service and event declarations from the `Booch` and `INT` tool configurations given below. The first declaration is a request that asks the process engine for permission to create a new library. The request is used in a condition in the interaction that implements the tool command for creating a new library. The second declaration is a specific service of the class interface editor for the creation of an inheritance relationship between two C++ class interface definitions. It is invoked from the Booch editor as soon as a user creates an inheritance relationship in the diagram and then the relationship will also be reflected in the affected C++ class interface definitions.

```
TOOL CONFIGURATION Booch;
 EVENT CrLibraryRequest(name:STRING):ERROR;
...
END CONFIGURATION Booch.

TOOL CONFIGURATION INT; ...
 SERVICE CrInheritance(inh_from : STRING;
                    visibility : STRING;
                    virtual: BOOLEAN):ERROR;
...
END CONFIGURATION INT.
```

The GTSL language and, in particular, its events and services are therefore powerful facilities for specifying tools that are customised for use with particular process models. They accomplish process automation at a much finer degree of granularity than the tool envelopes suggested by [Valetto and Kaiser, 1995], because multiple events and services can be exchanged during the execution of a tool. Tool envelopes only initiate the startup of a tool and obtain the result of its execution.

4.2.3 Architecture of Tools

The GTSL compiler GENESIS generates tools that store the documents they work upon as composite objects in the O_2 ODBMS. Therefore the GTSL compiler translates GTSL specifications into O_2 schemas that implement the structure and the operations available for objects representing abstract syntax graph nodes. A high-level overview of the tool architecture is displayed in Figure 2. Rectangles represent processes and arrows denote inter-process communication.

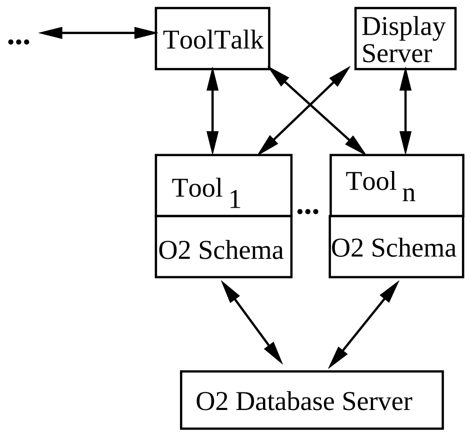


Figure 2: Architecture of Tools generated by GENESIS

Tools generated by GENESIS are integrated in several dimensions. They all store their documents in one central database server. The database is exploited for data integration, concurrency control and version management. Tools also communicate with a Display Server to implement group editing facilities so that users see modifications that concurrent users perform with a shared document version. Finally, tool services are invoked via the ToolTalk message server. Likewise tools communicate events that they have detected to the external world using that message server.

4.3 SLANG and SPADE

SPADE (Software Process Analysis, Design and Enactment) is a project that was carried out at CEFRIEL and Politecnico di Milano for the purpose of developing a process modelling language — i.e. a language to describe software processes — and a computer-based environment supporting the execution of processes according to their models. Two major results of this project were the SLANG (Spade LANGuage) process modelling language and the SPADE-1 PSEE.

4.3.1 Specification of Process Models in SLANG

We restrict ourselves to a rather concise description of SLANG, as the main purpose of this paper is to describe the experience of using SLANG. A more detailed account of SLANG is provided in [Bandinelli et al., 1993a, Bandinelli et al., 1993b, Bandinelli et al., 1994].

SLANG is a high-level Petri net language that has been adapted for software process modelling.

Transitions (represented as rectangles) model elementary process actions. *Places* (represented as circles) record the process state by storing tokens. *Tokens* are distinguishable and have a type that is defined by an class contained in the *SLANG type hierarchy* (see Figure 3). *Arcs* connecting places with transitions and transitions with places determine the consumption and production of tokens when transitions fire. The overall SLANG net defining a process model is hierarchically structured into *activities*. At the top of the hierarchy there is a starting activity called root activity, an example of which is shown in Figure 5

Transitions: The execution of a transition is atomic, in the sense that no intermediate state of the transition execution is visible outside the transition. Each transition is associated with a guard and an action. The guard is a predicate indicating whether the transition is enabled to fire. The action specifies how output tokens are computed.

Places: A place defines a template for a persistent repository of tokens. Each place has a name, which is a unique identifier within the activity, and a type (which has to be a subtype of the predefined class `Token`). A place can only contain tokens of its class type (or of any of its subclasses).

Arcs: SLANG offers different kinds of arcs, with intuitive semantics: *normal arcs*, represented by solid lines, *read-only arcs* represented by dashed lines, and *overwrite arcs*, represented by lines with double headed arrows. Arcs can be labelled by a weight, indicating the number of tokens flowing along the arc at each transition firing. An asterisk, “*”, indicates that as many tokens as possible flow through the arc when the transition fires.

Tokens: Process data are represented by objects, instances of subclasses of Token. Therefore, each token is typed and carries structured information that can be accessed through the methods defined in the corresponding class. Tokens represent different kinds of process data in the process model, including process products and by-products (source code, executable code, design documents, specifications, etc.), resources and organisational process aspects, (such as roles, skills, human resources, computing resources, etc.), and process model and state (for example, activity definitions, activity instances, class definitions etc.). The latter feature allows SLANG to provide computational reflection, and therefore offers the means to build meta-models, i.e. to model the manipulation and evolution of the model itself [Bandinelli et al., 1993a].

SLANG type hierarchy: The SLANG class hierarchy contains a set of predefined, process-independent classes that are part of the language definition and cannot be modified. The classes that are part of a specific process model are represented by a subgraph of the generalisation hierarchy whose root is `ModelType` (see Figure 3). Since SPADE is built on the object-oriented database management system O_2 [Deux, 1991], it was a natural choice to define the SLANG type system after that provided by O_2.

An example class used in the BA model is class `Library` displayed in Figure 4. It has a `name` and contains a design document (`BDiag`) in the form of a Booch diagram. As different library configurations have to be managed `Library` inherits from `CMItem`, which provides attributes

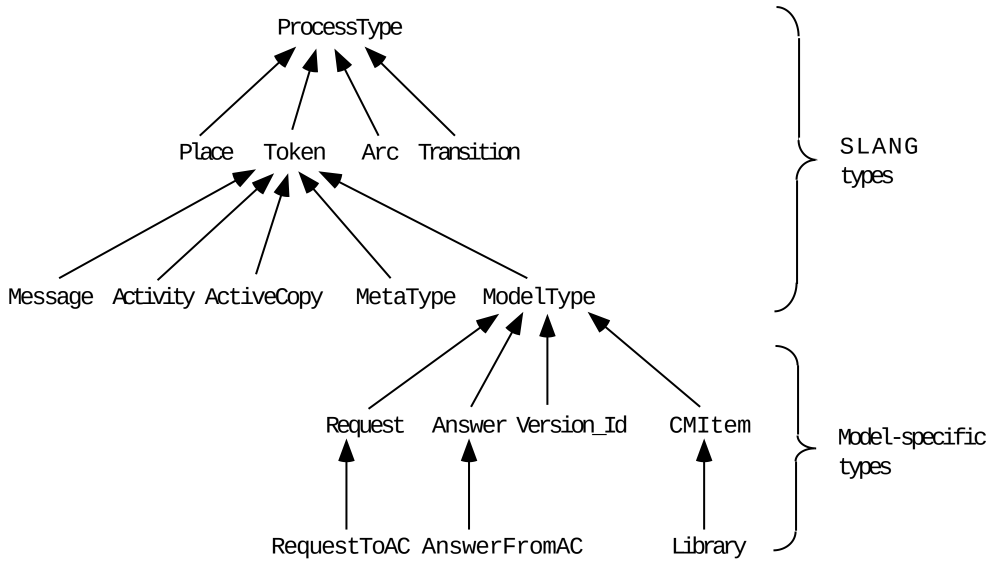


Figure 3: SLANG type hierarchy

and operations for manipulating configuration items, including revision and version numbers and methods to derive new revisions and versions.

```
class CMItem inherit ModelType
public type tuple (
          version:  integer,
          revision: integer,
          ...)
end
method body Derive_Version in class CItem {
  self->version +=1;
  self->revision = 0;
};
method body Derive_Revision in class CItem {
  self->revision +=1;
};

class Library inherit CMItem
public type tuple (
          name:  string,
          BDiag: BoochDiagr,
          ...)
end;
```

Figure 4: Example Type Definition

SLANG activities: Activities are the modularisation units provided by SLANG. An example showing activities is provided in Figures 5. According to the principles of information hiding, an activity definition has an *interface* and an *implementation* part. The activity interacts with other activities through its interface, while the implementation part remains hidden. An activity interface contains:

- A set of interface transitions, including *starting events* and *ending events*. Activity execution starts when any of the starting transitions fires and terminates with the firing of one of the ending transitions.

- A set of interface places, including *input places*, which play the role of arguments of an activity, *output places*, which contain the activity results, and *shared places*, whose contents can be used to exchange data with the calling activity.
- A set of interface arcs, connecting interface places and interface transitions.

Activity invocation is represented graphically by embedding an activity interface in a SLANG net. For instance, Figure 5 (showing the root activity of the BA model) includes invocations of activities `SessionManager`, `AccessControl`, `ConfManagement`, and `VersionManager`. When a starting transition fires, a new instance of the activity (called *active copy*) is created. Active copies are executed in parallel. For instance, two tokens in place `LoginMsg` would cause the starting transition of activity `Session Manager` to fire twice, thus creating two instances of that activity. These active copies are executed in parallel with the root activity. When an ending transition fires, it deposits the resulting tokens into some output places (e.g., `SessionManEnd`), belonging to the calling activity, then the terminated active copy is deleted (together with the tokens still contained, if any).

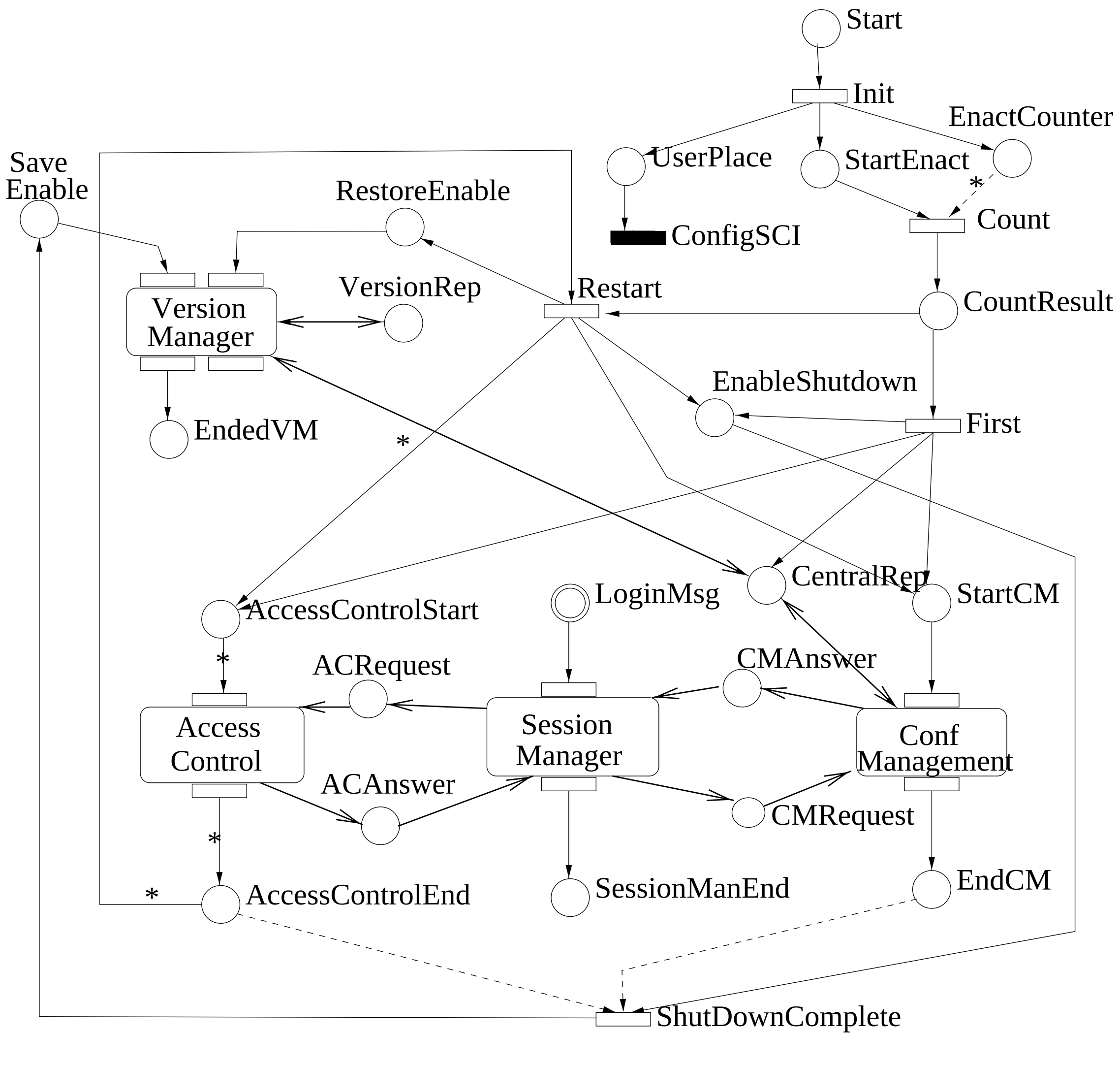


Figure 5: SLANG Net showing Root Activity of BA Process Model

4.3.2 Integration Facilities in SLANG

Besides *normal places*, SLANG includes *user places*. The contents of a normal place can change only because of a transition firing. User places, instead, change their contents as a consequence of events occurring in the user environment. When a process relevant event is generated (e.g., through a tool), the event is captured and a token with the information about the occurred event is inserted in the user place(s) associated with that event. Normal places are graphically represented by a single circle, user places are represented by double circles.

Events having effects only in the process model execution are modelled by means of regular (or white) transitions (represented by white rectangles). Events involving also effects in the user environment (e.g., launch a tool or change a tool state) are called *black transitions* and are represented as filled rectangles.

A black transition action contains the invocation of an external action (typically a tool service request). The action body is actually split in two distinct parts called *prologue* and *epilogue*. The prologue sets up the conditions for invoking the external action. In particular, the predefined variable `extAction` has to be given a string value, which indicates the external program to be executed, or specifies the service to be requested. If required, object parameters for the external action may be specified in the predefined variable `parList`. The epilogue contains the code to be executed after the external action termination. A predefined variable called `extResult` contains the results of the external action and together with the input values may be used to determine the transition output.

The execution of a black transition is not atomic: the prologue and the epilogue are executed in different moments, separated by the time necessary for the external action to complete. In the meanwhile, other transitions can fire.

4.3.3 Architecture of SPADE

This subsection provides an overview of the architecture of SPADE-1, the first full-fledged implementation of the SPADE PSEE. SPADE-1 is structured as indicated by Figure 6:

- The process enactment environment (PEE) contains the process engine and the object-oriented repository of the process-centred environment. The process engine is responsible for the execution of the SLANG process model.
- The user interaction environment (UIE) is composed of external tools contained in the development environment. The users (or process agents) interact with the process through the tools in the user environment.
- The SPADE communication interface (SCI) behaves as a communication bus, connecting SLANG interpreters in the PEE and tools in the UIE. The SCI also provides facilities for converting the communication protocol defined for the PEE into a specific protocol comprehensible by a given tool in the UIE.

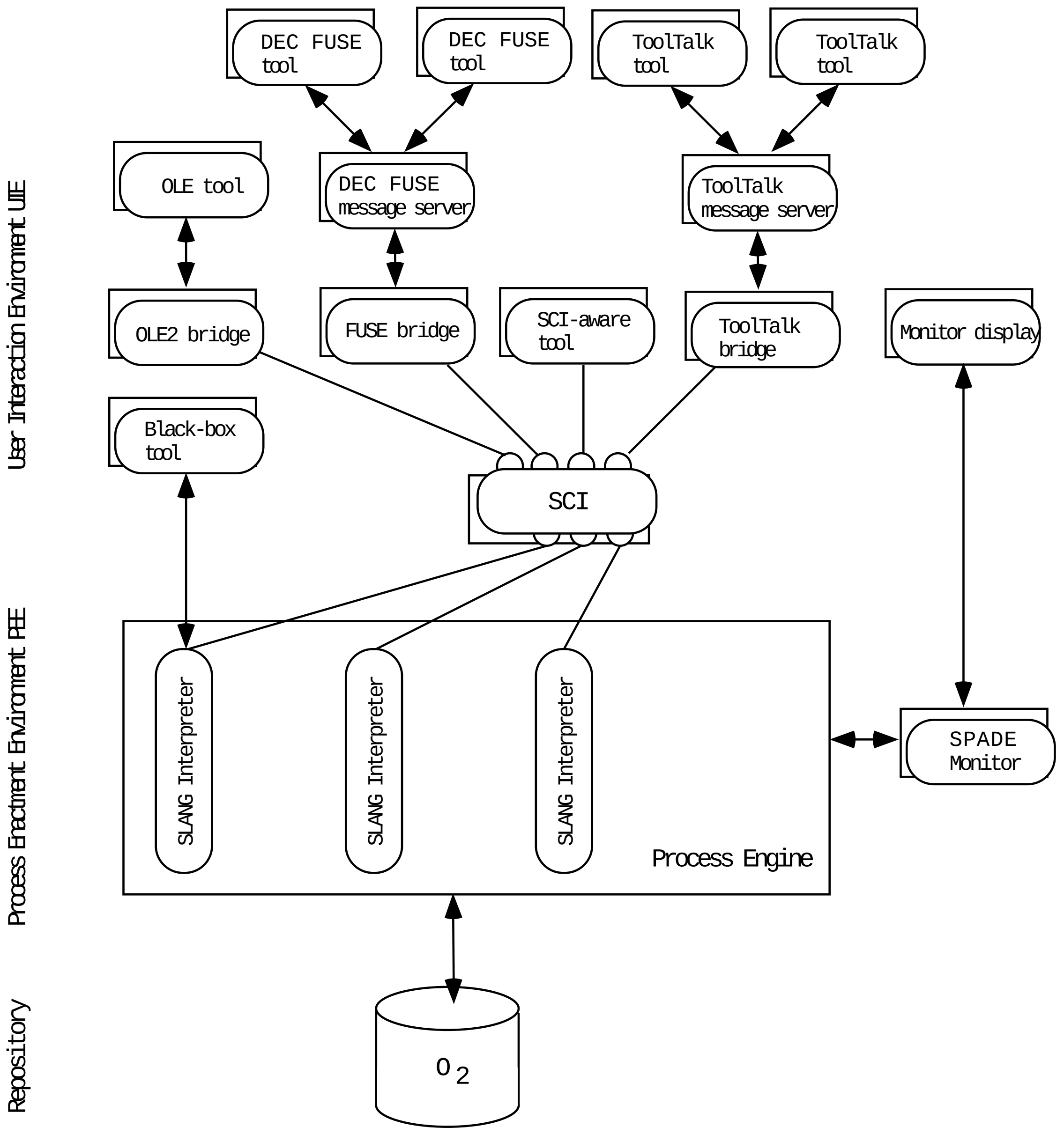


Figure 6: SPADE Architecture

Process Enactment Environment

The Process Enactment Environment is the run-time support that drives the execution of the process model. It is composed of a multi-threaded process engine and an O_2 object database [Deux, 1991]. O_2 has a client-server architecture where the server manages persistent storage (page management, concurrency control, etc.) and clients are in charge of computations (method execution, class or code compilation etc.). Process data (tokens) are stored in the O_2 data base, thus the SPADE process engine is an O_2 client, which uses the O_2 run-time system as the backbone for the execution of the SLANG interpreters. During process execution, each process engine executes one or more SLANG interpreter threads, each managing a set of active copies.

User Interaction Environment

The user interaction environment is composed of the tools used by process agents to perform their work. SPADE-1 provides a general mechanism to integrate these tools in the process environment. The level of integration that can be achieved depends essentially on the characteristics of the tool; tool generation facilities allow the construction of custom tools achieving the required degree of integration.

Tools are seen by the SPADE-1 environment as service-providers that can possibly share data with the process engine; a tool exhibits a control interface indicating the offered services and a data interface specifying the structure of used data. The differences among tools are basically given by the granularity of the offered services (the control integration dimension) and the level of agreement on the view of shared data (the data integration dimension).

With respect to the control dimension, tools can be classified into two broad classes: *black-box tools* and *service-based tools*. Black-box tools offer a single service that takes an input and produces some output. The action can be performed with or without user intervention, however there is no way to control the execution of the tool (there is only one service which can be requested, and the only event recognised by the process environment is the termination of the tool). Examples of black-box tools are UNIX programs such as *vi* and *cc*. Service-based tools provide several services, which can be requested individually. Integration of service-based tools can be obtained easily through the mechanism of message passing: tools send messages to a message server that forwards them to the recipient tools according to a given criterion. This mechanism, originally proposed by Reiss for the FIELD environment [Reiss, 1990] is becoming very popular and has been used in several commercial products (including DEC FUSE [Digital Equipment Corporation, 1992], HP SoftBench [Gerety, 1990], and SUN ToolTalk [Sun Microsystems, 1991]).

With respect to data integration, we also distinguish two broad classes: *file-system based* tools and *DB-system based* tools. File-system based tools are only able to exchange data with other tools in the form of files in the file-system. Each tool reads and writes data on files and is responsible for parsing and unparsing the data contained in the files. DB-system based tools use structured data stored in the database used by the process enactment environment. They are database clients and they share data definitions contained in the DB schema. In SPADE-1, DB-system based tools share class definitions in the O_2 database schema and are able to exchange database objects of arbitrary complexity and granularity.

SPADE Communication Interface

The SPADE Communication Interface is responsible for the communication between the user interaction environment and the process enactment environment. The global architecture of SPADE-1 is represented in Figure 6. The SCI acts as a communication server used by two kinds of clients:

- *PEE clients*: they are SLANG Interpreters that use the SCI as a means to communicate and interact with the "outside world", i.e., the UIE.
- *UIE clients*: they are tools in the UIE which provide services and notifications of relevant events to the SCI and (through the SCI) to the PEE.

Communication between the SCI, the tools and the process enactment environment is based on the message passing paradigm [Reiss, 1990] and follows the SCI Protocol. This protocol defines a connection procedure, an addressing mechanism and a message format. During the connection procedure SCI clients declare their type (UIE client or PEE client) and receive their address. Tools that are able to communicate through the SCI protocol are directly connected to the SCI, while other tools can be connected to the SCI through a sort of gateway (that we call *bridge*). The former tools are given an address consisting of a unique integer identifier, the latter are identified by a pair of integers: the first indicating the bridge, and the second indicating a tool in the set of tools connected to that bridge.

The message format defines different message types (requests, replies, notifications). Messages can contain only unstructured data. The SCI message format is modelled after those proposed by the commercial message-based tool environments[1].

PEE clients can request two kinds of services from the SCI: *configuration* and *integrated service-based tool invocation*. The configuration service allows SLANG interpreters to modify the behaviour of the SCI at enactment-time. For instance, a client can declare the kinds of messages that it is willing to receive; it has to specify a pattern (composed of a tool address and a message name) and will then receive the matching messages, as tokens, in a user place of the activity that issued the configuration request.

PEE clients can also ask for the invocation of an integrated service-based tool. The invoking SLANG interpreter provides the command to be executed (a command uniquely identifies a tool and a service) and the name of the host computer on which the tool must be executed. When the SCI receives an invocation request message from a SLANG Interpreter, it invokes the tool on the specified host and, in case of successful invocation, returns the tool identifier in the message reply, thus enabling further direct reference to the same tool.

A typical communication fragment is the following:

1. A process interpreter PI sends a configuration message to the SCI, indicating that it is willing to receive "SaveFile" messages arriving from tool with identifier T.
2. The user issues a SaveFile command through the editor T. The editor does not immediately execute the command: instead, it sends a message to the SCI indicating the request ("SaveFile") with its identity T.
3. The SCI uses its internal configuration tables to decide which process interpreter is interested in the request, then forwards the message to that interpreter. In this case the message is sent to interpreter P, and stored in a place whose name is "SaveFile".
4. The process environment processes the request (for example it checks the user's permission to modify the involved file). Then a message is sent to tool T containing the request to save the file or a warning to be displayed explaining why the file cannot be saved. Other messages can be sent through the SCI to people and tools interested in the file update.

[1]Actually, an open specification defining abstract, framework-neutral message interfaces for CASE tools was defined by Sunsoft, DEC, SGI and HP, and was submitted as a joint draft to ANSI X3H6 (recently renamed NCITS) and approved on January 21, 1997 with the name of CASE Tool Integration Messages. The SCI protocol was not designed to be compliant with the X3H6 proposal, however it would be relatively easy to make it compliant with the standard or to develop a X3H6 bridge.

5 The Experiment

The BA experiment is the first case study where the SPADE environment was used to model and enact an industrial-scale process model with the purpose of becoming the 'heart' of a PSEE. The experiment had a total duration of nine months starting in November 1994. Three parties participated: CEFRIEL, where the process model was developed, University of Dortmund, where development tools were generated and integrated into the BA SEE and BA's Infrastructure Group, from whom the process was elicited.

It is worthwhile noting that the skills of BA Infrastructure engineers were not sufficient to exploit SLANG nets for process modelling. The development effort was, therefore, mainly carried out full time by two Master students at CEFRIEL and University of Dortmund, who were supervised by experts in process modelling. When the development started, the students and their supervisors had a sufficient degree of knowledge of the languages to be used.

In this section, we discuss the experiment we conducted at British Airways from different perspectives. The first subsection discusses the elicitation and modelling of an enactable process model. The process model identifies a number of document types. The tools to be generated, therefore, have to include tools for the production of these document types. The generation of tools is discussed in the second subsection. The third subsection discusses the integration of processes and tools. In each subsection we explain the approach taken during the experiment and the results obtained.

5.1 Process Elicitation and Modelling

5.1.1 Approach

The starting point for process modelling was the BA process handbook. It provided an appropriate scenario from which other information regarding roles, responsibilities and library structure could be elicited during the first (kick-off) meeting. From there, process elicitation and modelling proceeded in a parallel and incremental way.

We started the formalisation of the process in the kick-off meeting using state transition diagrams, a notation Infrastructure members were familiar with. The result of this discussion is shown in Figure 7. It was during the development of this state transition diagram that the two participating Infrastructure members discovered the misunderstanding mentioned in Section 4.

The modelling activity continued by enriching the state chart with other context information regarding roles, library attributes, access restrictions etc. The state transition diagram was then formalised in a complete process program written in SLANG and in a protocol specification for the messages exchanged with tools through the SCI.

An important result that evolved from the development of this state transition diagram was the requirement that one developer should be responsible for a complete library and that only one developer at a time should work on a library. This implied that the granularity of documents considered by the process model had to be at the level of libraries rather than individual classes.

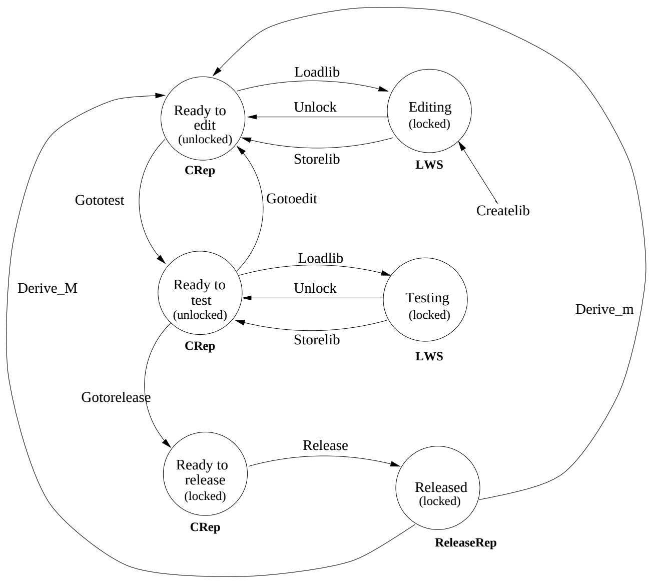


Figure 7: State Transitions during Library Management

5.1.2 Results

The whole process model developed for the experiment consists of five activities, the root activity (displayed in Figure 5), plus four other activities invoked by the root.

One of the activities invoked by the root activity is `SessionManager`, whose definition is displayed as the net shown in Figure 8. Whenever a user starts the Booch editor, the editor will notify a start-up event to the process model, which will appear as a token on place `LoginMsg`. This token enables `StartSM`, which will eventually fire, thus creating a new active copy of activity `SessionManager`. A component of the token identifies the user who has logged into the environment and this component is fired on place `Owner`. From there on, the activity awaits messages from the user interaction environment. These will appear as tokens on places `MsgFromSS` or `MsgFromBE`. After the net has done some consistency checks on the message and approved the message, the message tokens will appear on place `UserMsg`. Consequently, transitions are in conflict over these tokens and guards are used to determine the transition that fires. `DispatchToCM`, for instance, fires if the guard identifies the message as a configuration management message and transfers it to the place `CMRequest` from where it is consumed by the `ConfManagement` activity.

The SLANG net modelling the BA process is structured into five activities, containing in all 50 places and 65 transitions. The graphical net topology covers five pages in printed form. The size of the textual representation that includes the code for all token types and the different messages is about 4 KLOC.

Although the size of the model is fairly small it took a Master student, who had undergone training on the notation and its use for process modelling before he started, four months to complete the model. This effort might seem high at first glance. It has to be noted, however, that the effort was needed for process elicitation, formalisation of a logical process model, transformation of that logical model into a process program that integrates tools at a fine granularity, and the testing of the integrated environment.

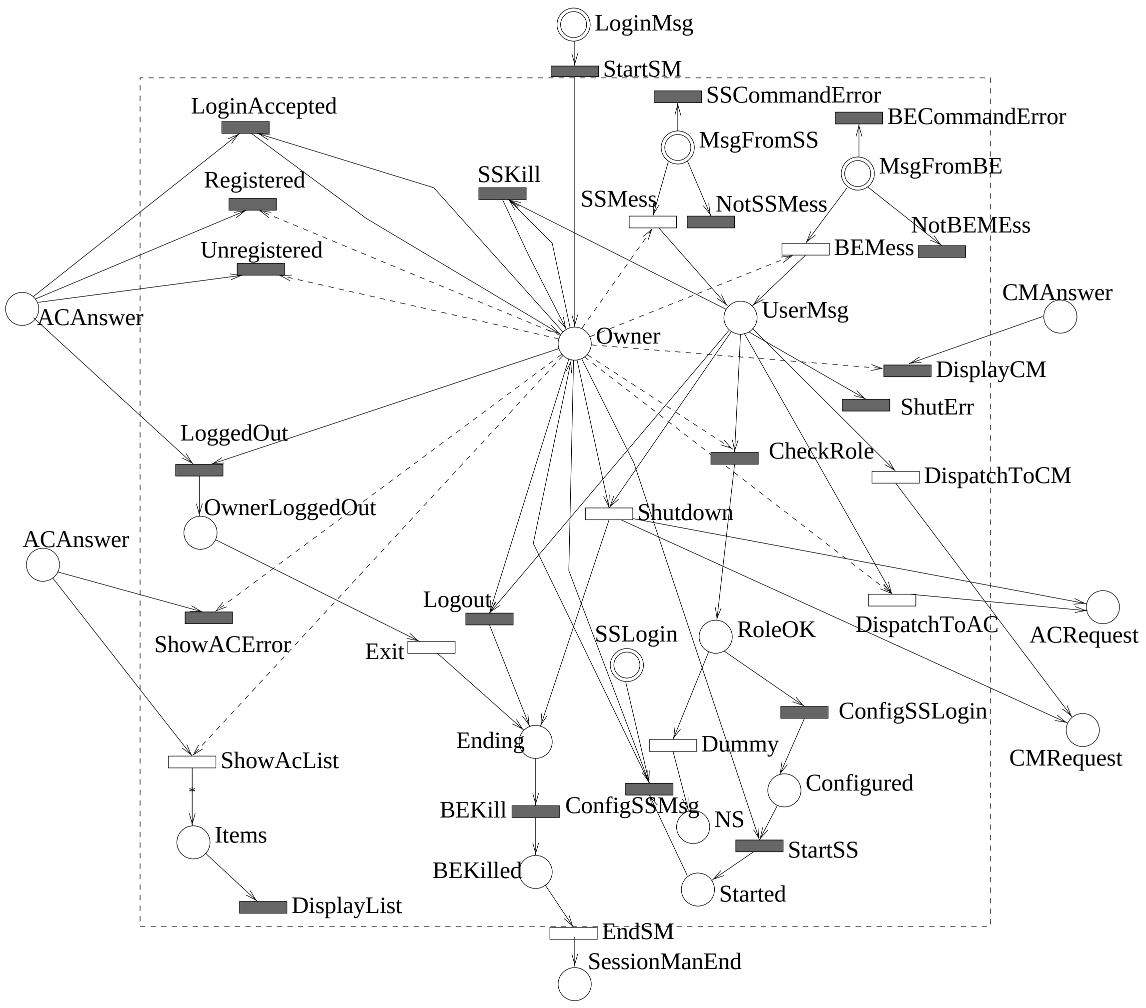


Figure 8: Refinement of Activity `SessionManager` (Figure 5)

We doubt that the effort would be considerably smaller with any other process programming language. Hence, the lesson to be learned is that coding enactable process programs that integrate tools at a fine level of granularity is expensive and that the cost effectiveness of fine grained process modelling whose aim is to derive an enactable process program has yet to be proved.

5.2 Tool Specification

5.2.1 Approach

The process elicitation exercises revealed that integrated tools for one graphical document type (Booch diagrams) and three textual document types (C++ class interface definitions, C++ class implementations and class documentations) would be needed. These tools were engineered in a syntax-driven way using GTSL. We started the specification of each tool by defining the context-free syntax. Extended and normalised BNFs (ENBNFs) were used for the languages supported by the textual tools.

An initial GTSL class hierarchy was derived from these syntax specifications. For each of the textual tools, the ENBNF syntax definition was translated into a hierarchy of GTSL classes using a similar strategy as in the interpreter design pattern discussed in [Gamma et al., 1993]. This translation was automated by an ENBNF compiler we generated for that purpose.

The generated GTSL classes were then elaborated further in order to specify the relevant static semantics rules and inter-document consistency constraints. These were expressed in a rule-based formalism from which the GTSL compiler created an incremental rule interpreter in a way discussed in [Emmerich et al., 1995].

We then added tool commands to each class for editing, analysing and browsing through documents. Editing commands were specified for the expansion of non-terminal placeholders into templates containing concrete lexemes and nested placeholders and the expansion of terminal placeholders into identifiers. In addition, editing commands for free textual input were specified. Analysis commands allow users to find usages of any declaration in order to allow change impact analysis. Browsing commands support users to open and locate documents based on semantic relationships between identifiers.

5.2.2 Results

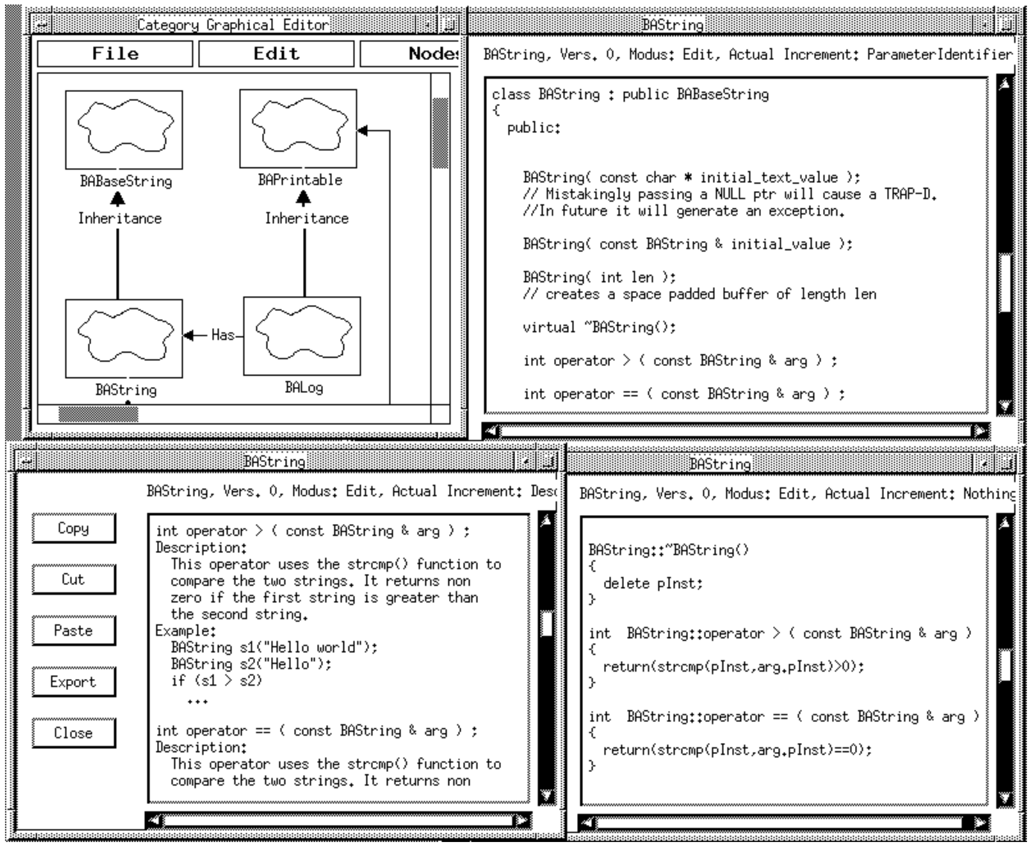
The user interface of the BA SEE provides four types of windows (shown in Figure 9). These windows are for the four different tools of the SEE: the Booch class diagram editor (upper left), the C++ class interface editor (upper right), the C++ method implementation editor (lower right) and the class documentation editor (lower left).

The Booch class diagram editor enables users to decompose libraries hierarchically into *categories* and *classes*. A category is a set of related classes and/or nested categories. Top-level categories represent libraries. Different types of relationships are supported to facilitate inheritance, aggregation and reference relations between classes. The Booch editor was specified using 20 classes.

The C++ class interface editor supports syntax-directed editing of C++ class definitions. The editor includes structure-oriented and free textual editing facilities that enforce syntactic correctness of class definitions. Moreover, the editor checks for correctness of the C++ static semantics while the user edits and visualises errors by underlining. Checking is done incrementally and transparently to the user. The C++ class interface editor has been specified in a way that it is integrated with the Booch editor so that, for instance, the creation of a new relationship in the Booch class diagram is reflected automatically in the C++ class interface definition. The class interface tool is specified using 75 GTSL classes.

The C++ method implementation editor supports programming of methods, which have been identified during the C++ class interface design. This editor is integrated with the C++ class interface editor in a way that any changes to, for instance, a method signature are automatically reflected in the implementation. We needed 130 GTSL classes to specify the C++ method implementation tool.

The class documentation editor implements the British Airways documentation standard, which requires a description for any method within a class together with an example of its application. The editor is also integrated with the C++ class interface editor, so that stubs are generated for each method identified in the class interface and users only have to complete



BAString, Vers. 0, Modus: Edit, Actual Increment: ParameterIdentifier

```
class BAString : public BABaseString
{
  public:


    BAString( const char * initial_text_value );
    // Mistakingly passing a NULL ptr will cause a TRAP-D.
    //In future it will generate an exception.

    BAString( const BAString & initial_value );

    BAString( int len );
    // creates a space padded buffer of length len

    virtual ~BAString();

    int operator > ( const BAString & arg ) ;

    int operator == ( const BAString & arg ) ;
```

BAString, Vers. 0, Modus: Edit, Actual Increment: Desc

```
int operator > ( const BAString & arg ) ;
Description:
  This operator uses the strcmp() function to
  compare the two strings. It returns non
  zero if the first string is greater than
  the second string.
Example:
  BAString s1("Hello world");
  BAString s2("Hello");
  if (s1 > s2)
    ...

int operator == ( const BAString & arg ) ;
Description:
  This operator uses the strcmp() function to
  compare the two strings. It returns non
```

BAString, Vers. 0, Modus: Edit, Actual Increment: Nothinc

```
BAString::~BAString()
{
  delete pInst;
}

int  BAString::operator > ( const BAString & arg )
{
  return(strcmp(pInst,arg.pInst)>0);
}

int  BAString::operator == ( const BAString & arg )
{
  return(strcmp(pInst,arg.pInst)==0);
}
```

Figure 9: User Interface of BA SEE Tools

these stubs. OS/2 IPF hypertexts and HTML files are generated from class documentations without requiring further actions of the developer. These two representations enable users of a class library to access documentation as on-line help facilities with standard browsers, i.e. without having to use the BA SEE. The documentation tool was easiest to specify and it needed only six GTSL classes.

A research assistant exclusively working on the project needed four months altogether to specify the classes of the four tools. The average size of a class specification was approximately 170 LOC and we extensively used inheritance to avoid duplication of properties.

5.3 Integration of Tools and Process

5.3.1 Approach

Specification of the integration Process-sensitive events, such as the creation of a new library, are captured by tools and the process model needs to be informed about them. This means that, from a modelling perspective, GTSL events are associated with SLANG user input places. Likewise, the invocation of tool services of a particular tool has to be expressed in the process model. During modelling this means that some of the black transitions are associated with GTSL services.

From an architectural point of view, the SCI has been successfully employed for implementing the association of GTSL events with user input places as well as for implementing black transition occurrences that invoke GTSL services. An event declared in the Booch editor specification is transformed into a Sun ToolTalk message, which is then sent via the ToolTalk bridge to the SCI. The SCI transforms the message into tokens, which include the message data, and user input places are marked with these tokens. Vice versa, a black transition causes the SCI to create a message that is then translated into a Sun Tooltalk message created by the ToolTalk bridge. Receipt of the message by the Booch editor causes the GTSL run-time system to invoke the service that is associated with the message.

The tools and the process model have to agree on some common message protocol. This involves syntactic and semantic concerns. From a syntactic point of view, tools and process model have to agree on the messages and their parameters. From a semantic point of view the meaning of messages and their parameters must be defined as well as the actions that they cause.

Architecture of the integration We considered different architectures for the integration of SPADE's PEE with the UIE. In particular, the decisions we had to make are:

- Whether to exploit the SCI as a software bus for the exchange of messages among tools, or to use another bus (e.g., ToolTalk).
- Whether to integrate the PEE with all the tools or with just a subset of them, leaving the integration of the others to be done "outside" the process.

In the case of the BA experiment these possible choices determined three different architectural alternatives.

The first alternative, which we refer to as the "Star" architecture, was to link all tools to the SCI, as shown in Figure 10. This choice means that the design relies on the SCI as the software bus supporting message exchange among the four BA tools.

The star architecture was rejected because it unnecessarily complicated the process model. In fact the criteria to be used in handling messages should have been coded in the process model. This is not a problem in general, since the PEE has to know messages that represent events in the UIE in order to react properly. In our specific case, however, the PEE is interested only in messages concerning the management of whole libraries, not data (like class interfaces) at a finer granularity. In other words, the interface, implementation and documentation editor could be integrated in a static way among themselves, without involving the process environment.

We refer to the second alternative as the "mediated architecture". This architecture exploits an observation we made during the initial process elicitation, i.e. that the granularity of responsibility at British Airways are class libraries rather than individual classes. As class libraries are defined in the Booch Editor, there is no need for integrating the editors for the other three document types with the process model. We, therefore, initially integrated the Booch editor directly with the SCI. The interface editor, implementation editor and documentation editor were locally integrated with the Booch editor through a ToolTalk based message server, as shown in Figure 11.

After the ToolTalk bridge was released the final "bridged architecture" was established, as shown in Figure 12. In this case, all the messages flowing among the tools are observed by the

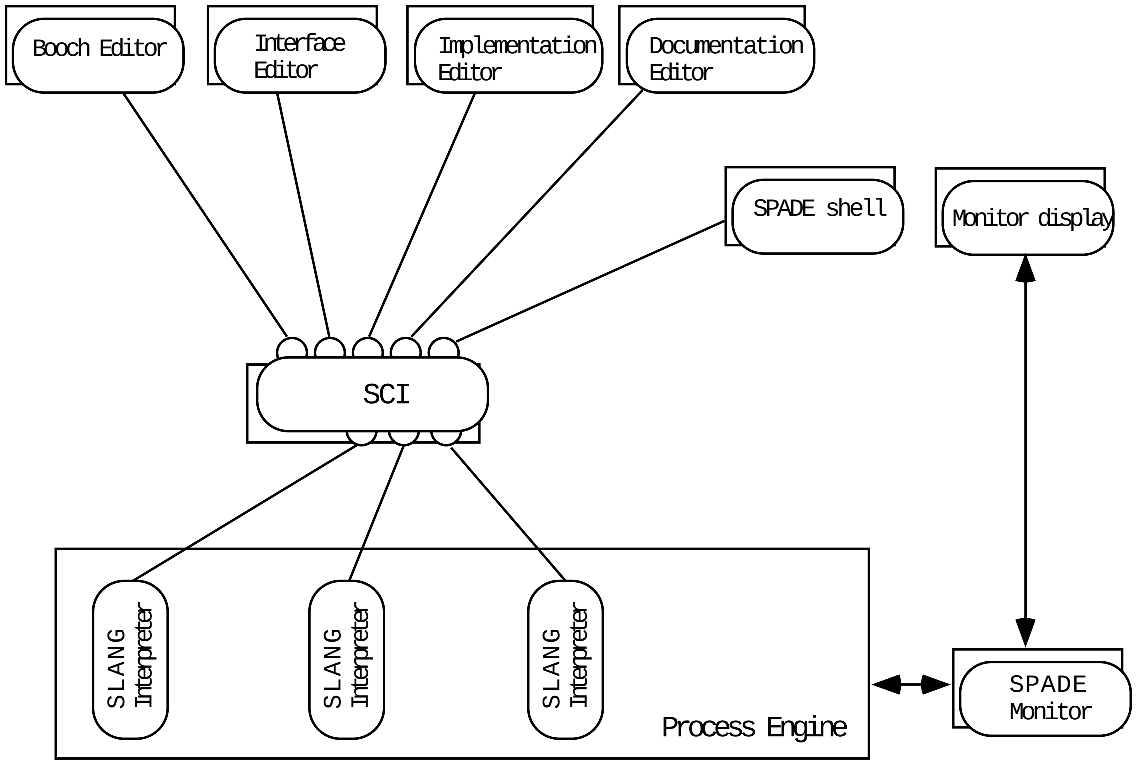


Figure 10: “Star” architecture

bridge – which is itself a ToolTalk tool – but only those important to the process are forwarded through the SCI to the PEE. Note that in this last case the design of the tools is cleaner, since they do not need to take into consideration the interaction with the SCI. In fact, they are not even aware of its existence. This is obviously obtained at the expense of building the bridge, which is however independent from the specific tools and process, and can therefore be reused as it is in other applications.

The last task in the definition of the model was to develop an appropriate communication protocol between the process engine and the Booch editor. A first version of this protocol took into account only services for session management and access control. Then the protocol was extended to capture the semantics of class library management at BA and the first enactable process model prototype was demonstrated in a meeting with BA Infrastructure members. The availability of an enactable prototype at this meeting proved very useful and triggered valuable feedback from Infrastructure members.

5.3.2 Results

For the British Airways process model, an informal protocol specification has been defined that includes 22 different messages.

An example is the `create` message in Figure 13. Messages like this are sent from the Booch class diagram editor via the ToolTalk bridge to the SCI and then appear as tokens on user input places. Likewise SLANG black transitions send messages to the Booch editor to invoke services.

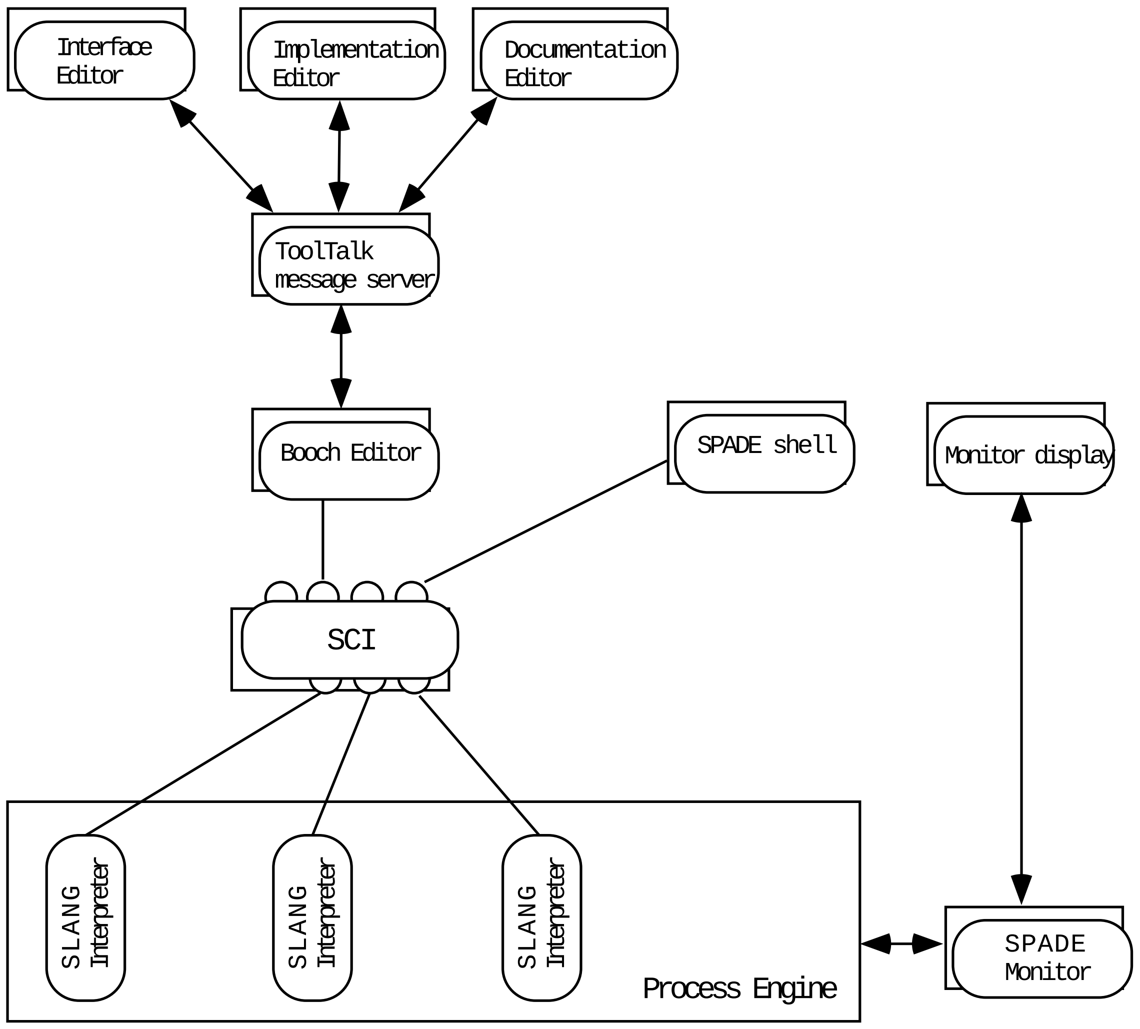


Figure 11: Mediated Architecture initially adopted in the BA Experiment

6 Lessons Learned

This section summarises the lessons learned from the perspective of both the technology provider and the technology user.

6.1 Experience of Technology Provider

The technology provider's concern was to assess the process modelling and automation capabilities of both SLANG and its support environment SPADE.

Modelling communication with tools. The process modelling language was deliberately designed to have two very simple and powerful means for modelling communication between process and tools: black transitions and user places. These two basic mechanism made it feasible to model virtually any interaction policy, provided an appropriate message protocol had previously been agreed.

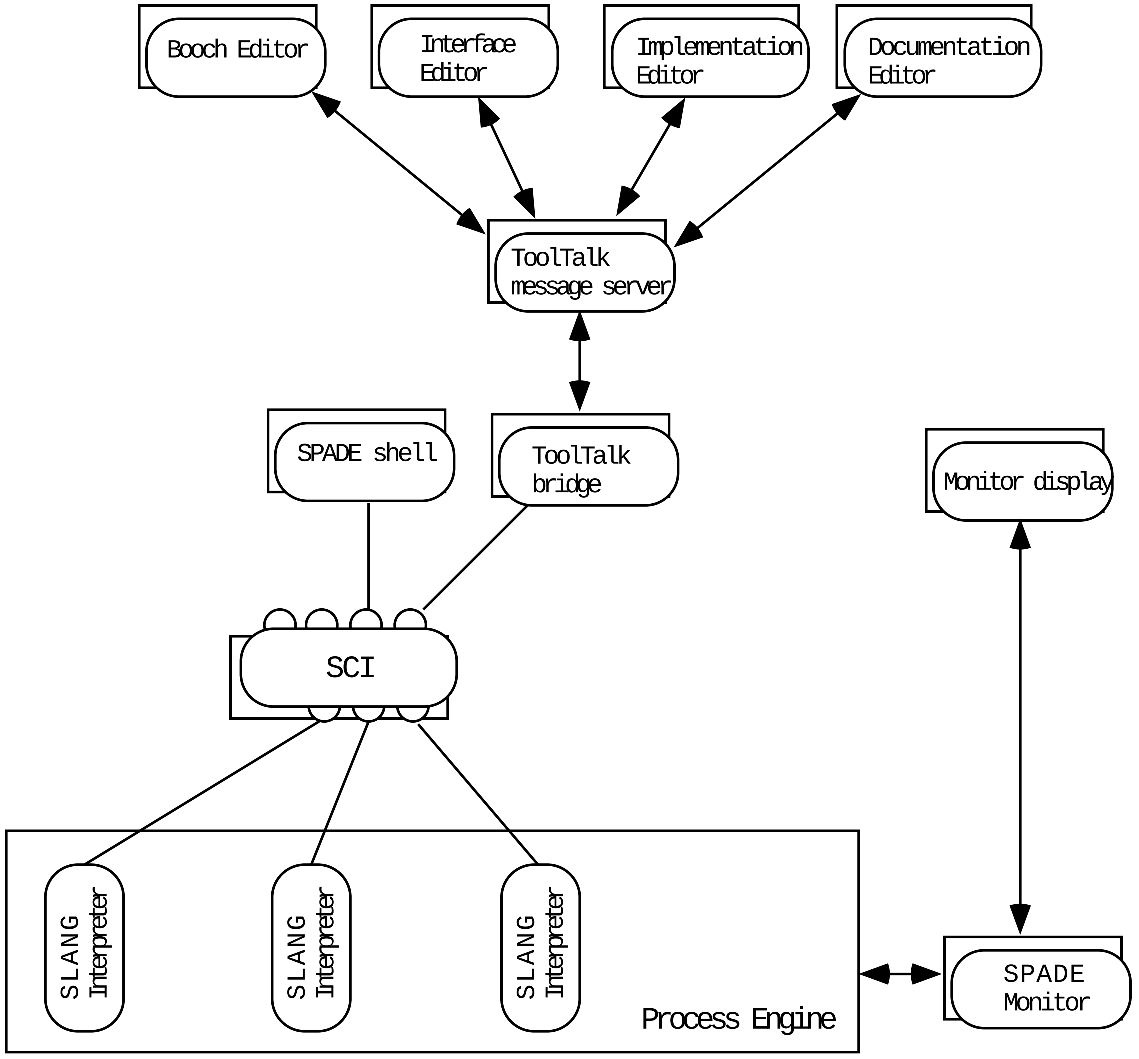


Figure 12: SPADE Architecture

```
Syntax:     create(lib_name:string)
Parameters: lib_name is the name of the library
            to be created
Return:     OK: library <lib_name> created
            ERROR: library <lib_name> already
                   exists
            ERROR: agent is not a librarian
```

Figure 13: Example Message Definition

Managing complex policies. The BA experiment showed that process policies can reach high levels of complexity in industry. Policy implementation has an impact on tools, on the communication protocol between tools and on the process model fragment managing that protocol. To avoid dealing with this additional complexity at the process level, higher level communication primitives should be introduced in the process modelling language. The trade-off, however, might be that the introduction of new communication primitives could complicate the current simple communication mechanism.

Abstraction mechanisms and activity interfaces. SLANG provides abstraction mechanisms that facilitate the hiding of complexity at lower levels of the process model. The activity `SessionManager` (see Figure 8), for instance, is completely independent from the access policy, which is hidden in the `AccessControl` activity (see Figure 5) and in the definition of owner data. In other circumstances, however, relevant behavioural information remains hidden in the activity implementation, too. The next transition that will fire in `SessionManager` after `DispatchToAC` has occurred will either be `LoginAccepted`, or `Registered`, `Unregistered` or `LoggedOut`. This, however, is not evident from just reviewing the `SessionManager` activity or the root activity. To resolve this problem, activity interfaces should not only be defined from a structural viewpoint, but should also provide a behavioural perspective. Such a perspective would have to show, at the activity invocation level, that marking `ACRequest` leads to a token at `ACAnswer` and that marking `CMRequest` produces a token on `CMAnswer`.

People who were not involved in the process modelling experiment did not find the enactable process model easy to understand. During process elicitation it was necessary to use simple state transition diagrams (like the one shown in Figure 7) to communicate with process stakeholders. Actually, this did not surprise us, since SLANG was initially conceived to support process enactment. The BA experience showed, however, that it is important to provide different views, which may require different levels of abstraction, different ways of structuring the process model and different formalisms to express them. This evidence supports the claims of [Deiters, 1993, Junkermann, 1995]. However, there are no satisfactory answers as yet to the problem of maintaining consistency of views during both modelling and enactment.

Performance issues. The enactment experience has shown that, during process model execution, the process engine is idle for most of the time. The engine works as a reactive system, which wakes up whenever a process-relevant event occurs. This observation seems to support the argument that performance issues are not relevant in process engines. However, fine-grained tool integration implies that the process engine becomes active during interactions of users with tools. To avoid low user acceptance of such an environment, response times less than a second have to be achieved. Being a pre-commercial prototype, the BA SEE has very reasonable performance characteristics with response times between 0.5 and 2 seconds on a SparcStation-20. However, the environment would need to be re-engineered to be used as a commercial development environment in industrial applications. Much of the overhead introduced in the process engine execution is due to the evolution mechanisms provided by the SPADE environment.

6.2 Experience of Technology Users

Process understanding. Much process understanding was gained through the process modelling process. The use of a formal process modelling language has lead to the removal of inconsistencies in the "Official Process", represented by Infrastructure's process handbook, and in the "Observed Process" [Bandinelli et al., 1995b], represented by what process agents thought the actual process was. In addition, the modelling activity had to be very detailed in order to obtain an enactable process representation. This facilitated discussions on solid grounds and prompted the removal of ambiguities in the process model.

Process Improvement. The process modelling activity highlighted flaws in the process, thus offering an opportunity to improve the process. In some cases, the introduction of process technology naturally leads to changing the process during modelling. To a certain degree, this process improvement was achieved by the BA experiment. However, to reduce the inherent risk it is advisable to apply process technology to already stable and well understood processes.

Process Automation. Although process automation has initially been perceived as a fascinating idea, not all processes are amenable to complete automation. Automated processes are less adaptable to unforeseen situations and, as pointed out above, their execution may introduce some performance overhead. Thus, while established administrative processes are better suited to process automation, less clearly defined creative processes can only be supported by, for example, providing guidelines or help information on demand.

Reference for future procurements. The British Airways Infrastructure Group has gained insights from this process elicitation and modelling exercise into the state-of-the art in process technology and software engineering environments. These insights have led them to adopt the BA SEE as a reference point for functionality expected from off-the-shelf software engineering tools.

7 Future work

The BA experiment highlighted several issues that are currently the subject of several research initiatives. This section describes these issues, and relates them to our current and future work.

7.1 The Industrial Context of the Experiment

The industrial context plays a fundamental role in the level of success that can be achieved in an experiment such as the one reported in this paper. In order to highlight the relevance of the industrial context, we briefly compare this experiment with a previous process improvement experiment in which the same process modelling language (SLANG) was used. The previous experiment was carried out at the Business Unit Telecommunications for Defense (BUTD) of Italtel, a large Italian telecommunication company [Bandinelli et al., 1995b]. Here are the main differences:

- *The experiment objectives.* The main objective of the Italtel experiment was to enhance the level of process understanding by reconciling the process agents' view of the process and the official process definition, represented by the Quality Manual. The British Airways experiment was focused on providing process support and, where possible, process automation. In this latter case, an improved process understanding was also obtained as a side effect.
- *The scope of the experiment.* Consistently with the experiment objectives, at Italtel BUTD only the process model was developed. Although it was a formal model, it did not include details, which are necessary for process enactment, such as integration with development tools. The formal SLANG process model served to call attention to some deficiencies in the process definition. At BA Infrastructure the SLANG process definition needed to be enriched with the appropriate code for tool integration so that the model could be enacted by the process engine.
- *The organisations' process maturity.* Being in the defence domain, Italtel BUTD must comply with security and quality control procedures. It is periodically assessed for

compliance with international standards, such as NATO AQAP-13, MIL-STD 2167A, and ISO 9000-3. Thus, there was a pre-existing process quality culture in BUTD. The Infrastructure Group at BA did not have this process culture and processes were only documented in brief before the experiment.

- *Integration of the experiment in a broader process improvement action.* The improvement of the Quality Manual of Italtel BUTD was part of a continuous improvement action and the results of the experiment served as a feedback to actually improve Italtel's Quality Manual. The experiment at BA Infrastructure did not have a direct link to any internal improvement action and although the experiment served to raise the level of understanding of some development processes and the environment was installed, it was not used in actual development.

This comparison shows that the industrial context in which a process improvement experiment is carried out is a decisive factor for its success. Specifically, the definition, development, tailoring and introduction of process technology cannot be addressed in isolation from a broader process improvement programme, having the necessary management commitment.

During recent years process improvement models (such as CMM [Humphrey, 1989], ISO-15504 (also known as SPICE) [ISO/IEC, 1997b], Boostrap [Kuvaja et al., 1994], etc.) and process technology have developed in independent ways. They address complementary aspects of process improvement and need each other support integrated process improvement. Process technology needs to become more mature to be able to support highly mature software organisations and improvement models need to rely on process technology in order to make process improvement more efficient and effective. Synergy between these two approaches is not fully explored yet and deserves further research and experimentation.

7.2 Process Life Cycle

The life cycle of a process resembles a software development cycle. The first step, as for any other software, should be to develop the process requirements. A specific process requirement language could be used to elicit process information from the various process agents in order to detect inconsistencies and misunderstandings among them at an early state. Languages that are simple and easy to understand, such as state charts, might be used during this phase. It might also be necessary to reconcile different process perspectives elicited from different process stake holders. That reconciliation should then determine a consistent logical model of the software process.

The transition from the logical software process model to a process program that is used for enacting development is not fully understood. We believe that the logical model and the process program are quite different entities. The process program has to address issues, such as tool integration and distributed execution, which should not be taken into account during process analysis. The result of the initial process programming stage should be a *process architecture* of the enacted process. It has to identify the main building blocks of the process program, the tools involved and the primitives and protocols used for integrating process and tools.

The stage of the software process development process that has gained most attention is *process programming*. Most process languages explicitly support the definition of executable process models. Process support environments are centred around interpreters for these languages and support their development, enactment and evolution.

It is very likely that different formalisms are used for identifying these different perspectives of requirements, architecture and coding. In order to provide an incremental and intertwined perspective on the meta process and allow for traceability decisions taken during process analysis, process architecture and process programming it is important that these formalisms be integrated. Hence the problems are similar to those in the software process itself. It needs to be investigated, however, whether solutions that have been developed for products can also be applied to the process modelling itself.

7.3 Support for Process Deviations

One of the lessons that we have learned is that software engineers are reluctant to have themselves strait-jacketed by a strict process model indicating what they should do, and how and when they should do it. This has led us to believe that strict enforcement of prescribed processes is too restrictive and will not be accepted in industry. This problem leads to two separate considerations. Firstly, the process support style can be *pro-active* or *reactive*, the latter allowing more freedom in the development process and being, therefore, more easily accepted. Secondly, it might be appropriate to allow *deviations* from the ideal reference model of the process.

SLANG and SPADE can support the pro-active and reactive styles equally well. The degree to which SPADE process programs can react to events, however, is limited by the ability of the UIE to capture those events that should trigger process reactions. For instance, some integrated development environments (such as DEC FUSE) do not always request permission to perform an action, or they do not notify some events. In these cases, the process environment has limited visibility to the user environment and cannot always react to events occurring in the user environment. A solution to this problem has been attempted in Provence [Barghouti and Krishnamurthy, 1995]. Provence is built on top of a patched operating system that generates event data from operating system calls. As soon as a certain pattern of UNIX system calls is recognised the Marvel process engine is notified of an event. While this approach seems promising for environments that rely on the UNIX file system, it almost certainly fails for tools that are built on database systems, which might not use the operating system for secondary storage management.

In any kind of process, inconsistencies within documents are generally tolerated by the tools. Developers either explicitly perform checks to find inconsistencies or tools implicitly check and visualise inconsistencies. At specified points in time, for instance before baselining or translating a document, developers remove inconsistencies. Similar considerations might be applicable to the software process itself.

Process technology should, therefore, contribute techniques and tools that inform engineers how they are performing with respect to a prescribed process rather than restricting them to only those activities that are in-line with the process defined. Currently, we see two different approaches to tolerate inconsistencies on the process level. The modelling of *allowable deviations* and *compliance checks*.

The SENTINEL PSEE [Cugola et al., 1996] tolerates deviations during process enactment. SENTINEL process descriptions are composed of a behavioural and an intentional part. SEN-

TINEL supports process enactment according to its behavioural description, but it allows humans to deviate from that description, provided that the process is still in the boundaries established by the intentional description. During deviations the system keeps track of the progress of the process, registering all the state changes and the operations that are performed. When a reconciliation is needed, information on all the data that have been potentially "polluted" by the deviation – and that might need to be repaired – are calculated by reasoning on the deviation history.

In [Emmerich et al., 1997b] a process actually performed is said to be *compliant* if it conforms to the modelled process. An approach for checking compliance to process through inconsistency analysis between documents is suggested. The suggestion is based on the observation that standards for software processes, such as ISO-12207 [ISO/IEC, 1995] or the forthcoming ISO-15288 [ISO/IEC, 1997a], require relevant process information to be documented in management reports, minutes, progress reports and the like. Hence, compliance rules can be expressed in terms of relationships that must hold between these process state documents and the product documents. Object-oriented techniques to specify the document type structure and event-condition-action languages for the definition of compliance rules are being explored.

The problem of process deviation has been tackled by several researchers, who proposed different techniques to manage inconsistency. Wolf and Cook proposed compliance checks based on event trace comparison [Cook and Wolf, 1995], while in APPL/A [Sutton et al., 1994] exception handling is used to deal with exceptional situations.

7.4 Interoperability and Distribution in PSEE Architectures

In our experiment development of the message protocol for the integration of process model with tools took a very considerable amount of time. The most important reason for this was the absence of a high-level protocol specification against which the implementation in both the process model and the tools could be checked in an automated manner.

In general, the integration between tools and process engines faces problems that are common to a number of distributed systems. For instance, communication between tools and process synchronisation of messages has to be considered; concurrent messages might be arriving at the process engine from several tools; tools processes might not be available and may have to be launched; communication can be disturbed by slow or unavailable networks; and tools and process engine might be running on different platforms requiring heterogeneity in data representation to be resolved.

These problems are not new. They have been addressed in a number of distributed system infrastructures, for instance the Common Object Request Broker Architecture (CORBA) defined by the Object Management Group. Use of this architecture might be advantageous for the implementation of events and services that are used for communication between process and tools. Using CORBA, events and services can be implemented as parameterised synchronous, or deferred synchronous, or asynchronous operation execution requests, rather than as messages that are exchanged. The signatures of these operations are defined in an Interface Definition Language (IDL) rather than in untyped messages. Compliance of both processes and tools to the interfaces can be checked statically at compile time or dynamically at runtime. The different activation strategies included in the CORBA standard define how tools or process engine should respond to multiple concurrent communication requests. CORBA has been explicitly designed to support heterogeneous environments where components are implemented in different programming languages and can be running on different hardware and operating system platforms, hence resolving all the heterogeneity problems.

While CORBA seems to provide better support than message based systems for the integration of tools with process engines, a number of questions remain open. CORBA defines the lower level primitives for distributed operation invocation that we have sketched above; In addition the CORBAservices and CORBAfacilities specification also standardise solutions to higher-level problems that commonly occur in distributed systems. The problems addressed in these specifications include naming and trading, security, licensing, concurrency control, transaction management and event notification. It is interesting to identify those CORBAservices that are needed in a distributed PSEE architecture and to see whether the specified services are sufficient for PSEEs. It is quite likely that they are not sufficient and that they will have to be adapted to PSEE architectures. [Emmerich, 1996a, Emmerich, 1997] report on the results of initial investigations concerning the size of the gap between what higher-level services are needed in a PSEE architecture and the features that are actually defined in the CORBAservices and CORBAfacilities specifications.

7.5 Removing Barriers for Acceptance in Industry

Process technology has little popularity in the software industry. Among the causes of this limited acceptance are:

- Limited empirical evidence of the costs and benefits provided by the process-centred approach to software development, since precise, quantitative cost/benefit evaluations are still missing.
- Perceived risk of adoption, since process technology is relatively new and sophisticated.

As far as SLANG and SPADE are concerned, past industrial applications at the Italian telephone operator Italtel [Bandinelli et al., 1995b] and at BA were just experiments carried out with the objective of testing the technical features of the process modelling language and of the PSEE. What we need now is a small set of industrial trial applications concentrating on the definition of a path to process-centred development, including process modelling, deployment and usage.

This is exactly the objective of the ESPRIT Project 23768 DOOR (Developing Object Oriented applications Rapidly), which is currently in its initial phase[2]. DOOR aims at demonstrating the suitability and cost effectiveness of GOODSTEP technology to build industrial software applications. In particular, DOOR will use SLANG and SPADE in two different industrial contexts: the software production environment for automation systems of Mannesmann Datenverarbeitung, and the power system environment of ENEL (the major Italian utility company).

DOOR will define a process for modelling, deploying and using the GOODSTEP process technology. The technology will have to be integrated within the existing technical, managerial, organisational and cultural environment. This clearly implies the identification of possible risks and the corresponding risk management actions. Special attention will be devoted to the deployment plans, whose objectives will be:

[2]DOOR is carried out by a consortium including three organisations formerly belonging to the the GOODSTEP consortium, namely Engineering, the University of Frankfurt, and CEFRIEL.

- to define the scope of the experiment, i.e. to decide which parts of the process to model and enact first;
- to carry out a suitable training program because, for the first time, SLANG and SPADE will be used in an industrial environment by teams lacking any SLANG/SPADE expert, and initial training will be of crucial importance.

The cost of setting-up and running a process-centred software development environment will be measured, and the resulting benefits will be carefully assessed, at least on a qualitative basis.

8 Summary

We have reported on the experimental application of process technology at British Airways (BA). We used SLANG to model BA's C++ class library management process, and we constructed an experimental process-centred software engineering environment based on SPADE and incorporating tools specified in GTSL and developed by means of GENESIS.

In this experiment, SPADE was used for the first time to enact part of an industrial process. The experiment was successful, since it demonstrated that SPADE can actually support a real industrial process and as the environment is used for a reference point in future procurements at BA. Nevertheless, some issues were also highlighted that are not satisfactorily dealt with by PSEEs and therefore deserve additional research.

The existence of a process life-cycle was recognised. The process needs support in all its phases, which are much like those of software development. SPADE proved suitable for supporting the low-level design and coding phases, but the enactable process model was too detailed in itself to improve process understanding by the BA personnel.

Moreover, we recognised the need to support process deviations as engineers might have to deviate from the model if a situation arises that was unforeseen in the process model. Adapting the process model in these occasions is not really an alternative to deviations, given the effort it takes to implement and test a change.

Finally, from an architectural perspective the need was recognised for an investigation of how the integration of tools and process engines in a PSEE can be specified and coded at appropriate levels of abstraction. The message based approach chosen in this experiement was inappropriate.

Acknowledgements

We are indebted to a number of people. Alfonso Fuggetta, Carlo Ghezzi, Wilhelm Schäfer and Roberto Zicari provided the support within GOODSTEP to undertake the experiment we reported about. Alain Ainsworth pointed us to the problems of the BA Infrastructure Group. Mark Phoenix provided us with valuable insights into daily development practice at British Airways. Antonio Carzaniga and Giovanni Vigna made fundamental contributions to the implementation of SPADE-1. Riccardo Rodriguez refined the BA process with SLANG to a degree that it could be enacted. Jörg Brunsmann designed and implemented most of the integration between the Booch tool and the process model. Elisabetta Di Nitto, Anthony Finkelstein and Carlo Montangero explained the importance of deviations to us. Finally, Stephen Morris helped us to improve the presentation of this paper.

References

[Arlow et al., 1994] Arlow, J., Phoenix, M., and Pryce, B. (1994). The British Airways Application Scenario for GOODSTEP. Deliverable ESPRIT Project GOODSTEP 26P, Commission of the European Union, DG III.

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

[Bandinelli et al., 1995a] Bandinelli, S., Baresi, L., Fuggetta, A., and Lavazza, L. (1995a). Experiences in Implementation of a Process-centered Software Engineering Environment using Object-Oriented Technologies. *Theory and Practice of Object Systems (TAPOS)*, 1(2):115–131.

[Bandinelli et al., 1993a] Bandinelli, S., Fuggetta, A., and Ghezzi, C. (1993a). Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144.

[Bandinelli et al., 1994] Bandinelli, S., Fuggetta, A., Ghezzi, C., and Lavazza, L. (1994). SPADE: An Environment for Software Process Analysis, Design and Enactment. In Finkelstein, A., Kramer, J., and Nuseibeh, B., editors, *Advances in Software Process Technology*, pages 223–247. Research Study Press Limited.

[Bandinelli et al., 1993b] Bandinelli, S., Fuggetta, A., and Grigolli, S. (1993b). Process Modeling-in-the-large with SLANG. In *Proc. of the 2^{nd} Int. Conf. on the Software Process, Berlin, Germany*, pages 75–83. IEEE Computer Society Press.

[Bandinelli et al., 1995b] Bandinelli, S., Fuggetta, A., Lavazza, L., Loi, M., and Picco, G. P. (1995b). Modeling and Improving an Industrial Software Process. *IEEE Transactions on Software Engineering*, 21(5):440–454.

[Barghouti and Kaiser, 1990] Barghouti, N. S. and Kaiser, G. E. (1990). Multi-Agent Rule-Based Software Development Environments. In *Proc. of the 5^{th} Annual Knowledge-Based Software Assistant Conference*, pages 375–387.

[Barghouti and Krishnamurthy, 1995] Barghouti, N. S. and Krishnamurthy, B. (1995). Using Event Contexts and Matching Constraints to Monitor Software Processes. In *Proc. of the 17^{th} Int. Conf. on Software Engineering, Seattle, Washington*, pages 83–92. IEEE Computer Society Press.

[Cook and Wolf, 1995] Cook, J. E. and Wolf, A. L. (1995). Automating Process Discovery through Event-Data Analysis. In *Proc. of the 17^{th} Int. Conf. on Software Engineering, Seattle, Washington*, pages 73–92. ACM Press.

[Cugola et al., 1996] Cugola, G. P., Di Nitto, E., Fuggetta, A., and Ghezzi, C. (1996). A Framework for Formalizing Inconsistencies in Human-Centred Systems. *ACM Transactions on Software Engineering and Methodology*, 5(3).

[Deiters, 1993] Deiters, W. (1993). *A View-based Approach to Software Process Management.* PhD thesis, University of Dortmund, Dept. of Computer Science.

[Deiters and Gruhn, 1990] Deiters, W. and Gruhn, V. (1990). Managing Software Processes in MELMAC. *ACM SIGSOFT Software Engineering Notes*, 15(6):193–205. Proc. of the 4^{th} ACM SIGSOFT Symposium on Software Development Environments, Irvine, Cal.

[Delobel and Madec, 1993] Delobel, C. and Madec, J. (1993). Version Management in O_2. Technical report, O_2-Technology.

[Deux, 1991] Deux, O. (1991). The O_2 System. *Communications of the ACM*, 34(10).

[Digital Equipment Corporation, 1992] Digital Equipment Corporation (1992). *DEC-FUSE manual.*

[Dinkhoff et al., 1994] Dinkhoff, G., Gruhn, V., Saalmann, A., and Zielonka, M. (1994). Business Process Modeling in the Workflow Management Environment LEU. In Loucopoulos, P., editor, *Proc. of the* 13^{th} *Entity-Relationship Approach*, number 881 in Lecture Notes in Computer Science, pages 46–63. Springer.

[Emmerich, 1996a] Emmerich, W. (1996a). An Architecture for Viewpoint Environments based on OMG/CORBA. In Vidal, L., Finkelstein, A., Spanoudakis, G., and Wolf, A., editors, *Joint Proceedings of the SIGSOFT '96 Workshops*, pages 207–211. ACM Press.

[Emmerich, 1996b] Emmerich, W. (1996b). Tool Specification with GTSL. In *Proc. of the* 8^{th} *Int. Workshop on Software Specification and Design, Schloss Velen, Germany*, pages 26–35. IEEE Computer Society Press.

[Emmerich, 1997] Emmerich, W. (1997). CORBA and ODBMSs in Viewpoint Development Environment Architectures. In Orlowska, M., editor, *Proc. of* 4^{th} *Int. Conf. on Object-Oriented Information Systems, Brisbane, Australia.* Springer. To appear.

[Emmerich et al., 1997a] Emmerich, W., Arlow, J., Madec, J., and Phoenix, M. (1997a). Tool Construction for the British Airways SEE with the O_2 ODBMS. *Theory and Practice of Object Systems.* To appear.

[Emmerich et al., 1997b] Emmerich, W., Finkelstein, A., Montangero, C., and Stevens, R. (1997b). Standards Compliant Software Development. In *ICSE Workshop on Living with Inconsistency, Boston.*

[Emmerich and Gruhn, 1991] Emmerich, W. and Gruhn, V. (1991). FUNSOFT Nets: A Petri-Net based Software Process Modeling Language. In *Proc. of the* 6^{th} *Int. Workshop on Software Specification and Design, Como, Italy*, pages 175–184. IEEE Computer Society Press.

[Emmerich et al., 1995] Emmerich, W., Jahnke, J.-H., and Schäfer, W. (1995). Object Oriented Specification and Incremental Evaluation of Static Semantic Constraints. Technical Report 24, ESPRIT-III Project GOODSTEP, http://www.dbis.informatik.uni-frankfurt.de/REPORTS/GOODSTEP/GoodStepReport024.ps.gz.

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

[Emmerich et al., 1993b] Emmerich, W., Schäfer, W., and Welsh, J. (1993b). Databases for Software Engineering Environments — The Goal has not yet been attained. In Sommerville, I. and Paul, M., 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.

[Engels and Groenewegen, 1994] Engels, G. and Groenewegen, L. (1994). SOCCA: Specifications of Coordinated and Cooperative Activities. In Finkelstein, A., Kramer, J., and Nuseibeh, B., editors, *Software Process Modelling and Technology*, pages 71–102. Research Studies Press, Tanton, UK.

[Ferrandina et al., 1994] Ferrandina, F., Meyer, T., and Zicari, R. (1994). Implementing Lazy Database Updates for an Object Database System. In *Proc. of the* 20th *Int. Conference on Very Large Databases, Santiago, Chile*, pages 261–272.

[Ferrandina et al., 1995] Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., and Madec, J. (1995). Schema and Database Evolution in the O_2 Object Database System. In *Proc. of the* 21th *Int. Conference on Very Large Databases, Zürich, Switzerland*, pages 170–181.

[Gamma et al., 1993] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1993). Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Nierstrasz, O., editor, *ECOOP '93 — Proc. of the* 7th *European Conf. on Object-Oriented Programming, Kaiserslautern, Germany*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer.

[Gerety, 1990] Gerety, C. (1990). HP SoftBench: a new generation of Software Development Tools. *HP journal.*

[GOODSTEP Team, 1994] GOODSTEP Team (1994). The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In Ohmaki, K., editor, *Proc. of the Asia-Pacific Software Engineering Conference, Tokyo, Japan*, pages 410–420. IEEE Computer Society Press.

[Humphrey, 1989] Humphrey, W. (1989). *Managing the Software Process.* Addison Wesley.

[ISO/IEC, 1995] ISO/IEC (1995). *International Standard, Information Technology Software Life Cycle Process. ISO 12207.*

[ISO/IEC, 1997a] ISO/IEC (1997a). *Draft Systems Engineering Standard. ISO 15288.* To appear.

[ISO/IEC, 1997b] ISO/IEC (1997b). *Software Process Improvement and Capability dEtermination.* International Standardisation Organisation. To appear.

[Junkermann, 1995] Junkermann, G. (1995). A Dedicated Process Design Language based on EER-Models, Statecharts and Tables. In *Proc. of the* 7th *Int. Conf. on Software Engineering and Knowledge Engineering, Rockville, Maryland*, pages 487–496. Knowledge Systems Institute.

[Kuvaja et al., 1994] Kuvaja, P. J., Simila, J., Krzanik, L., Bicego, A., Saukkonen, S., and Koch, G. (1994). *Software Process Assessment and Improvement – The Bootstrap Approach.* Blackwell, Oxford, United Kingdom.

[Maier, 1989] Maier, D. (1989). Making Database Systems Fast Enough for CAD Applications. In Kim, W. and Lochovsky, F. H., editors, *Object-Oriented Concepts, Databases and Applications*, pages 573–582. Addison Wesley.

[Peuschel and Schäfer, 1992] Peuschel, B. and Schäfer, W. (1992). Concepts and Implementation of a Rule-based Process Engine. In *Proc. of the* 14th *Int. Conf. on Software Engineering, Melbourne, Australia*, pages 262–279. IEEE Computer Society Press.

[Peuschel et al., 1992] Peuschel, B., Schäfer, W., and Wolf, S. (1992). A Knowledge-based Software Development Environment Supporting Cooperative Work. *International Journal for Software Engineering and Knowledge Engineering*, 2(1):79–106.

[Reiss, 1990] Reiss, S. (1990). Connecting Tools using Message Passing in the FIELD Program Development Environment. *IEEE Software*, pages 57–67.

[Sun Microsystems, 1991] Sun Microsystems (1991). *Solaris/Moracsa Open Windows: The ToolTalk Service.* Sun MicroSystems, Inc.

[Sutton et al., 1994] Sutton, S., Heimbigner, D., and Osterweil, L. (1994). APPL/A: A Language for Software Process Programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286.

[Sutton et al., 1990] Sutton, S. M., Heimbigner, D., and Osterweil, L. (1990). Language Constructs for Managing Change in Process-Centred Environments. *ACM SIGSOFT Software Engineering Notes*, 15(6):206–217. Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, Cal.

[Valetto and Kaiser, 1995] Valetto, G. and Kaiser, G. (1995). Enveloping "Persistent" Tools for a Process-Centred Environment. In Schäfer, W., editor, *Proc. of the 4th European Workshop on Software Process Technology, Nordwijkerhout, The Netherlands*, volume 913 of *Lecture Notes in Computer Science*, pages 200–204. Springer.