CORE
Metadata, citation and similar papers at core.ac.uk
Provided by UCL Discovery

Fine grained Process Modelling: An Experiment at British Airways*

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

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

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

Jim Arlow
8a Butler Ave.
Harrow
Middx, HA1 4EH, UK
100010.401@compuserve.com

Abstract

We report 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 (PSEE) based on SPADE. BA required processes to be automated at a finer degree of granularity than 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 with 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 representation for humans trying to understand the process model. A more comprehensible notation is needed for design and documentation purposes. Although the airline did not deploy the PSEE in its production environment, the experiment proved beneficial for BA. The modelling uncovered serious flaws in the existing process, and the BA engineers improved their knowledge of process technology.

*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) and J. Arlow was with British Airways (UK).

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 basis. Examples of these languages are extensions to programming languages (e.g. extensions of Ada [24] and Prolog [21]), Petri net based approaches (FUNSOFT [15] and SLANG [4]) and multi-paradigm approaches integrating several high-level descriptions (such as ESCAPE [18]). Process modelling environments have been constructed for these languages so as to edit, simulate, analyse, enact and evolve process models. Examples include the Merlin environment [20], the SPADE environment [2], Melmac [10] or Marvel [6]. 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. We refer to such an environment as a *process-centred software engineering environment* (PSEE).

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 [12] 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. British Airways is a large software developer in the UK, with some 2,000 IT staff. An increasing number of operations research programs (such as fleet or crew allocation) are now being developed with object-oriented techniques. 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. It 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 from a high-level specification written in GTSL (the GOODSTEP Tool Specification Language). An account of this tool construction effort is given in a companion paper [14]. 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.

The paper is structured as follows. Section 2 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 4 presents the way the process modelling experiment was conducted. Section 5 summarises the experiment results. We then conclude with a summary of the lessons that we have learned from this process modelling experiment in Section 6.

2. Goals of the Experiment

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

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

Feasibility: Is it feasible with the language primitives available in SLANG and GTSL to decrease 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 performance of a fine-grained process enactment appropriate 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 it had reached a common (i.e., consistent) understanding of the process, after having worked for two years on the maintenance of class libraries.

Process Improvement: The group knew flaws of 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.

3. Baseline of the Experiment

3.1. 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.

Two main problems were identified by Infrastructure 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." [1]*

An obvious approach to enforcing the change control procedures is to model them with a process modelling language and to guide the process by enacting the model. This requires development tools to be integrated into the process environment in a way that tools cannot be abused to circumvent the process that has been modelled. Moreover, tool integration, at least partially, has to be fine-grained so that the process model can react to the execution of individual tool commands, rather than complete tool sessions. The reason is that there are usually documents, such as Booch diagrams, whose components represent other documents. Inter-document consistency constraints require the creation, modification or deletion of documents whenever components are created, modified or deleted. Whenever a new class icon is created in a Booch diagram, for instance, a corresponding class interface definition document has to be created. Since 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 in one document that represents another document has 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.

3.2. Tool Specification and Generation

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 [14]. While the basic concepts of GTSL have been presented in [13], language primitives for the description of fine-grained process integration of GTSL based tools are illustrated in Section 3.2.2, after a brief presentation of the functionality of tools to be integrated with the process model.

3.2.1 BA SEE Tools

The Booch class diagram editor, whose user interface is shown in the upper left corner of Figure 1, 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 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 C++ method implementation editor supports programming of methods, which have been identified during the C++ class interface design. This editor is

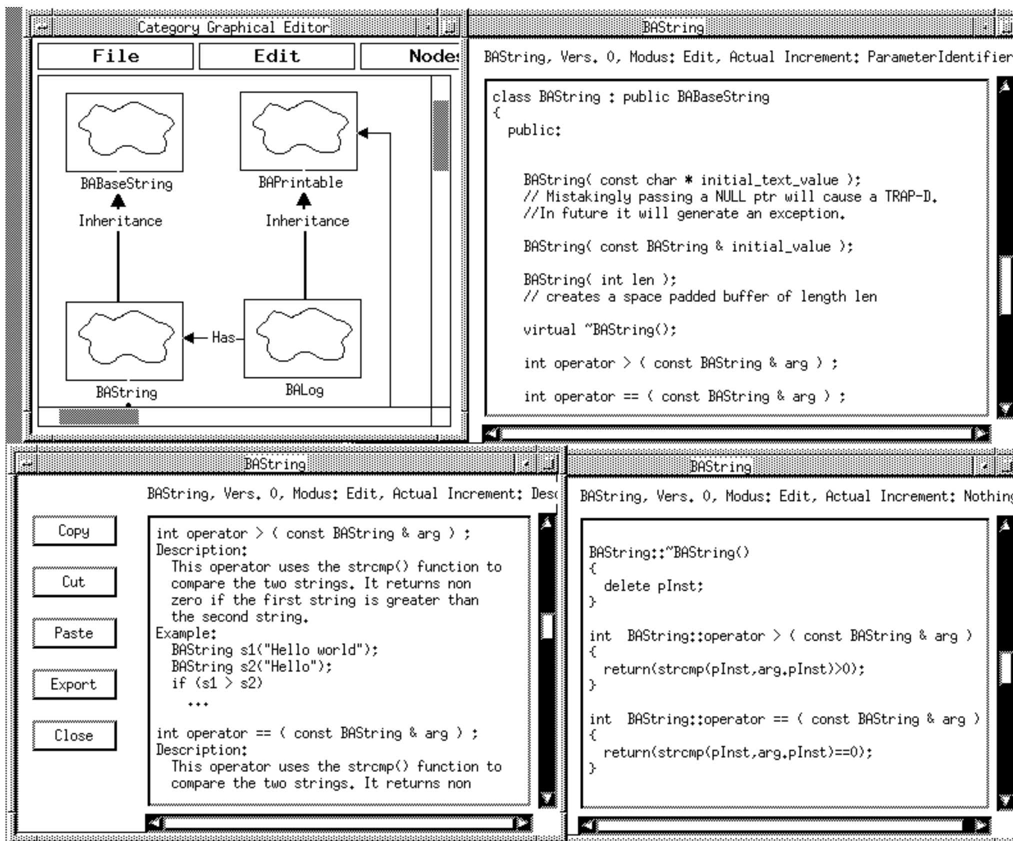

Figure 1. User Interface of BA SEE Tools

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.

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 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.

3.2.2 Integration Facilities in GTSL

GTSL specifications of environments use the object-oriented paradigm to define the tools and documents, which are internally represented as project-wide abstract syntax graphs [16]. An environment specification is structured into a number of *tool configurations*. Each of these consists of a number of *classes* that define the different node types that occur in project-wide abstract syntax graphs. Different *sections* are provided to define properties of a class such as attributes, abstract syntax and semantic relationships. The available operations to modify graph nodes are defined in a *method section*. The invocation of operations from commands and their availability are specified as patterns in *interaction sections*. *Multiple inheritance* enables properties from super classes to be reused in subclasses.

Tools generated from GTSL specifications can not only be used through a user interface, but 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 *generic* and *tool-specific services*. GTSL defines some 20 generic services, examples of which are the creation of a document, the 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 have to be specified by the tool builder. They are declared in tool configura-

tions [13] and implemented in the same style as GTSL interactions.

Events allow the tool to inform the environment about certain incidents as opposed to services, which enable the environment to communicate with a tool. An event can either be a *request* or a *notification*. A request is sent to a consumer, who 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 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 as 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, events and services are therefore powerful facilities to specify tools that are customised for use with particular process models. They accomplish process automation at a much finer degree of granularity than tool envelopes [25], because multiple events and services can be exchanged during the execution of a tool.

3.3. SPADE and SLANG

SPADE (Software Process Analysis, Design and Enactment) is a project that was carried out at CEFRIEL and Politecnico di Milano. Two results of this project were the SLANG (Spade LANGuage) process modelling language and its implementation in SPADE-1. We do not explain SLANG and SPADE here and assume that the reader is familiar with the concepts. A language reference manual is available in [7]. Detailed descriptions of SLANG can also be found in [2, 4, 3]. In the next subsection we describe how SLANG was used to model the BA process.

3.3.1 SLANG

Use of SLANG Activities

SLANG allows complex process to be decomposed into a structured set of fairly independent *activities*. The activity construct was also used to model the BA process. Figure 2 presents the overall process model (the *root* activity), which contains sub-activities `SessionManager`, `AccessControl`, `ConfManagement`, and `VersionManager`, each modeling a specific sub-process. 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.

Activity invocations are represented graphically by embedding an activity interface into a SLANG net. Activity interfaces are provided with interface transitions, i.e., *starting transitions* (placed on the upper side of the activity box) and *ending transitions* (on the lower side). When a starting transition fires, a new instance of the activity (called *active copy*) is created. Active copies are executed in parallel; this feature was used to model the concurrent work of BA engineers. For instance, two tokens in place `LoginMsg` 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. An activity can also exchange data with the calling activity through *shared places*. For instance, the `SessionManager` sends `CMRequests` to the root activity and receives `CMAnswers`.

White transitions represent elementary steps of processes. Their execution 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 or not. The action specifies how output tokens are computed. An example is the `Restart` transition in the root activity of the BA model.

Use of SLANG Integration Facilities

Besides *normal places*, SLANG includes *user places*. The contents of normal places are changed by transitions, while changes on user places are triggered by

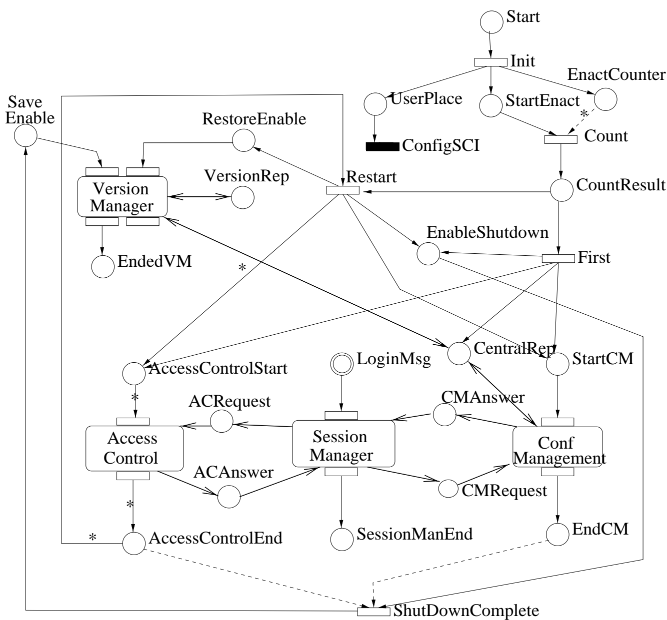

Figure 2. Root Activity

external events. Normal places are graphically represented by a single circle, user places are represented by double circles. User places have been exploited to model communication of events between tools and process model. Whenever a Booch editor is started, for instance, it creates a login message token on the user input place `LoginMsg`.

Process steps involving effects in the user environment (e.g., launching a tool or changing a tool state) are called *black transitions*. They are represented as filled rectangles. An example is the `ConfigSCI` transition that configures the SPADE Communication Interface to route messages in a given way (see below). Black transitions are also exploited in the BA process model for the definition of the invocation of tool services.

3.3.2 SPADE

Process Enactment Environment

The Process Enactment Environment is the bottom layer of the SPADE architecture displayed in Figure 3[1]. It is the run-time support that drives the execution of the BA process model. It is composed of a multi-threaded process engine (PE) based on the O_2 object database [11]. Process data (tokens) are stored in the O_2 database, 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 SLANG interpreters. During process execution, each process engine executes one or more SLANG interpreter threads, each managing a set of active copies.

[1] Figure 3 reports only the components used in the BA model. The current implementation of SPADE offers also bridges for DEC FUSE and OLE2 tools, in addition to the possibility of invoking black-box tools.

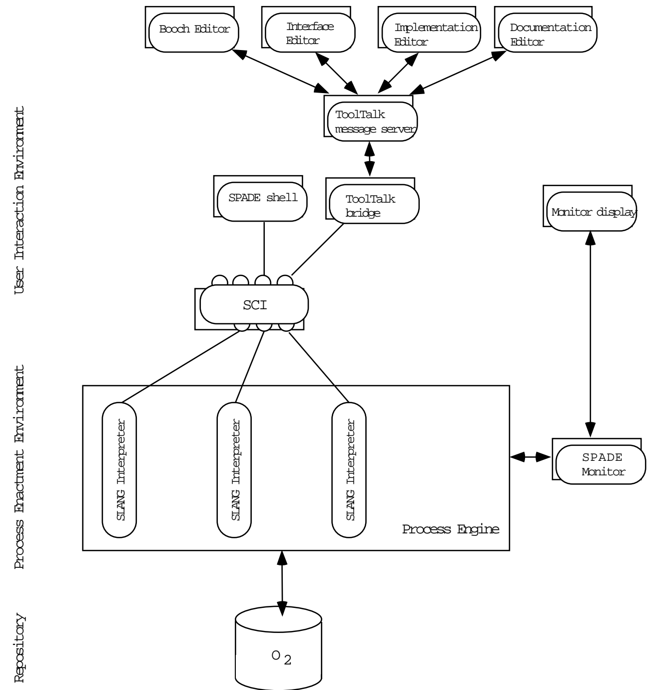

Figure 3. Customisation of SPADE

User Interaction Environment

The user interaction environment is composed of the tools used by process agents to perform their work. 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 differences of the granularity between the offered services.

Tools can be classified into two classes: *black-box tools* and *service-based tools*. Black-box tools offer a single service that takes an input and produces some output. There is no way to control the execution of the tool. Examples of black-box tools are UNIX programs such as *vi* and *cc*. Service-based tools provide several services, each of 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 for-

wards them to the recipient tools according to a given strategy. This mechanism, originally proposed by Reiss for the FIELD environment [22] is becoming very popular and has been used in several commercial products (including DEC FUSE [8], HP SoftBench [17], and SUN ToolTalk [23]).

SPADE Communication Interface

The SPADE Communication Interface (SCI) is responsible for communication between the User Interaction Environment (UIE) and the Process Enactment Environment(PEE). Thus, the SCI provides facilities for converting the communication protocol defined for the PEE into a specific protocol tools in the UIE can understand. The SCI acts as a communication server used by two kinds of clients:

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

Communication between the SCI, tools and the process enactment environment is based on the message passing paradigm [22] and follows the SCI Protocol. 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 a *bridge*). Since GTSL generated tools send and receive Sun ToolTalk messages, the `ToolTalk bridge` is exploited in the BA SEE.

PEE clients can request 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.

4. 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 to note 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 modeling. When the development started, the students and their supervisors had a sufficient degree of knowledge of the languages used.

4.1. Process Elicitation and Modelling

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 4. It was during the development of this state transition diagram that the two participating Infrastructure members discovered the misunderstanding mentioned in Section 3.

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.

4.2. Architecture of the Integration

Different alternatives were proposed for the integration between process model and the development tools. For instance, an alternative was to link all tools to the SCI. This alternative was rejected because

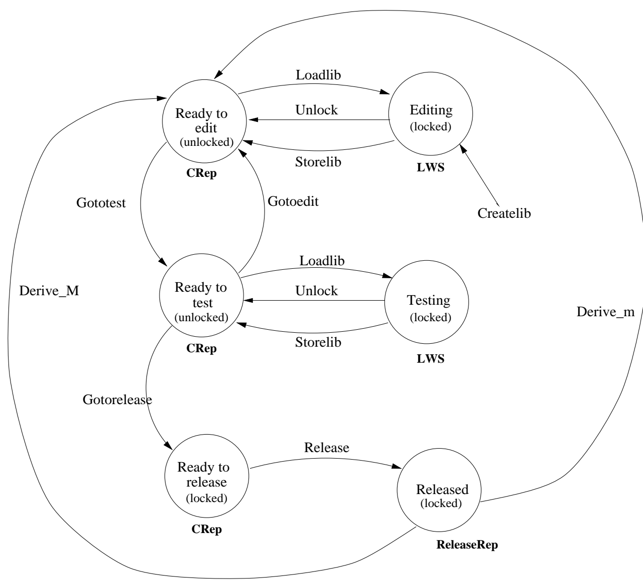

Figure 4. State Transitions during Library Management

it would have unnecessarily complicated the process model, since the process model needed only to manage 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.

Therefore the architecture that was finally adopted is based on the observation that only the Booch Editor tool needs to exchange control information with the process engine. Initially the Booch editor was directly integrated with the SCI. The interface editor, implementation editor and documentation editor were locally integrated with the Booch editor through a ToolTalk based message server. Later, when the ToolTalk bridge was released, the final architecture was established, as shown in Figure 3.

Based on this architecture, the next step in the modelling 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 in this meeting proved very useful and triggered valuable feedback from Infrastructure members.

5. Results

5.1. Integration of Tools and Process Program

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 services that is associated with the message.

5.2. Protocol Specification

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.

For the British Airways process model, an informal protocol specification has been defined that includes 22 different messages. An example is the `create` message below.

```
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
```

Messages like this are sent from the Booch class diagram editor to the SLANG process model to implement events. Likewise SLANG black transitions send messages to the Booch editor to invoke certain services.

5.3. The Process Model

The whole process model developed for the experiment consisits of five activities, the root activity, as displayed in Figure 2, plus other four activities invoked by the root. All the activities contain, in total, 50 places and 65 transitions. The translation of the graphical representation of the model into a textual representation generates a file of about 4 KLOC. This provides a quantitative idea of the model size.

One of the activities invoked by the root activity is `SessionManager`, whose definition is displayed as the net shown in Figure 5. 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.

`SessionManager` is a SLANG implementation of a command interpreter. Messages can be seen as command strings and the SLANG net checks the appropriateness of the command (for instance in transitions `BECommandError` and `NotBEMess`). The implementation of `SessionManager` required a considerable amount of time. This was because SLANG provides low-level concepts for expressing computations on the basis of O_2C, but it does not provide high-level constructs, such as ordered-attribute grammars [19], that would be more appropriate for specifying command interpreters. This caused the effect that even minor changes to the message protocol were quite time-consuming to implement.

The process model development has been directed towards efficient enactment of the BA software process and its integration with tools from the user interaction environment. During the process of its development, BA Infrastructure members learned quite a lot about their process. The process of eliciting and modelling the development process was probably more useful for understanding than its result. Unexperienced readers

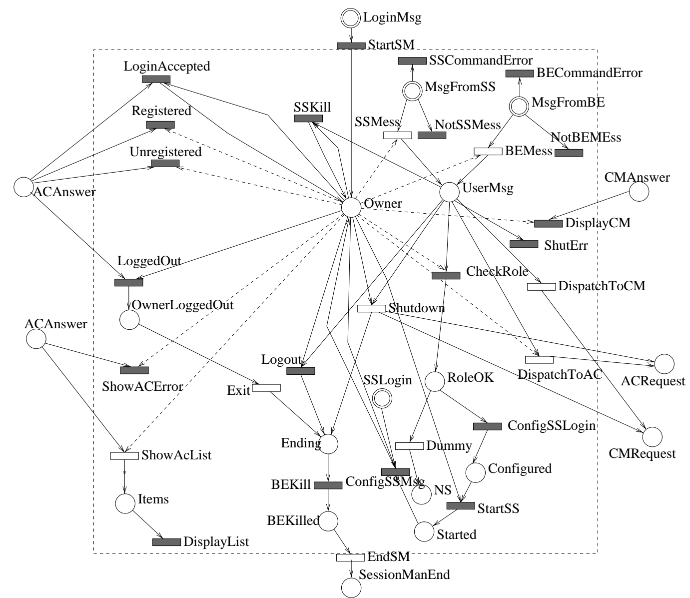

Figure 5. Refinement of Session Manager

will find it difficult to understand the SLANG model of the process. One reason is that activity definitions do not distinguish between net components that are critical to understanding the overall process (e.g. `CheckRole` in Figure 5) and those included for enacting the model (e.g. `Dummy`).

Decomposition of the overall net into activities complicates further an understanding of the model. Certain transitions that belong together from the point of view of control flow have been separated by decomposition into activities. 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`.

6. Lessons Learned

The main result of the experiment carried out at British Airways was our demonstration that process technology could reduce the gap between the process that was actually happening and the process described in the Infrastructure group's process handbook. This goal was achieved by modelling this process partially

and enacting it by a fine-grained integration of development tools and process engine. We now revisit the experiment goals presented in Section 2 and examine the lessons that we have learned.

6.1. Experience of Technology Provider

A first concern was to assess the expressiveness for process automation of both the process modelling concepts of SLANG and its support environment SPADE. The 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 is previously agreed.

Through the BA experiment, we learned how complex process policies can be in industry. Policy implementation has an impact on tools, on the communication protocol between tools and on the process model fragment managing that protocol. This introduced additional complexity to the already complex process model. SLANG provides abstraction mechanisms that facilitated the hiding of complexity at the lower levels of the process model. For instance, activity `SessionManager` is completely independent from the access policy, which is hidden in the `AccessControl` activity and in the definition of owner data. However, industrial scale process modelling also requires high-level, easily combinable, process-oriented abstractions or building blocks, which are missing in SLANG.

People who were not involved in the process modelling experiment are little able to understand the enactable process model. For instance, during process elicitation it was necessary to use simple state transition diagrams (as the one shown in Figure 4). Actually, this did not surprise us, since SLANG was initially conceived mainly 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 [9, 18]. However, there are no satisfactory answers yet as to how such views may be kept consistent during both modelling and enactment.

British Airways did not deploy the BA SEE developed in GOODSTEP. One of the reasons was that the impact of the deployment was too radical a change. The BA SEE contains completely new development tools, such as the Booch editor and the C++ class interface editor, and explicit process constraints, such as a library code cannot be modified if the corresponding Booch diagram is not modified first. In addition, the modelled processes were not fully understood and institutionalised at British Airways. The lessons that we have learned from this is that process technology adoption has to be done in an evolutionary rather than a revolutionary way and should focus on mature processes first. For the purpose of bringing process technology into industrial practice, it would have been more appropriate to start with partial process monitoring support which, for instance, provided context-sensitive process guidance on user demand, rather than support that attempted to completely control and automate the process. After some period of technology assimilation and process maturation, it might have been possible to introduce process automation and more advanced tools.

SLANG and SPADE would have supported this evolutionary introduction of process technology in several ways. Firstly, the introduction of process technology does not necessarily require the user interface to be changed. As a matter of fact, users can continue using tools they are familiar with without having to learn a new user interface. Secondly, the level of process enforcement is adjustable in the process model. Therefore, it can range from a simple process monitoring to complete process automation, depending on the characteristics of the organisation. Finally, the mechanisms available in SPADE for process evolution could be used to change the process model, even during its enactment.

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 time constraints below a second have to be met.

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

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" [5], represented by what process agents thought the actual process was. In addition, the modelling activity was accomplished in considerable detail in order to obtain an enactable process representation. This facilitated discussions on solid ground and prompted the removal of ambiguities in the process model.

The process modelling activity highlighted flaws in the process, 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 risk inherent to the introduction of process technology, it is advisable to apply it to already stable and well understood processes.

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.

Acknowledgements

Mark Phoenix provided us with valuable insights into daily development practice at British Airways. 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. We are indebted to Alain Ainsworth, for pointing us to the problems of the BA Infrastructure group, and to Alfonso Fuggetta, Carlo Ghezzi, Wilhelm Schäfer and Roberto Zicari for providing the support within GOODSTEP to undertake this effort. Finally, we thank Stephen Morris for comments that helped us to improve the presentation of this paper.

References

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

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

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

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

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

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

[7] Politecnico di Milano CEFRIEL. SLANG Process Modeling Language Reference Manual version 2.2. Deliverable ESPRIT Project GOODSTEP 8P, Commission of the European Union, December 1994.

[8] Digital Equipment Corporation. *DEC-FUSE Manual*, 1992.

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

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

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

[12] G. Dinkhoff, V. Gruhn, A. Saalmann, and M. Zielonka. Business Process Modeling in the

Workflow Management Environment LEU. In P. Loucopoulos, editor, *Proc. of the* 13th *Entity-Relationship Approach*, number 881 in Lecture Notes in Computer Science, pages 46–63. Springer, 1994.

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

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

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

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

[17] C. Gerety. HP SoftBench: a new generation of Software Development Tools. *HP Journal*, June 1990.

[18] G. Junkermann. 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, 1995.

[19] U. Kastens. Ordered Attributed Grammars. *Acta Informatica*, 13(3):229–256, 1980.

[20] B. Peuschel and W. Schäfer. 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, 1992.

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

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

[23] Sun MicroSystems, Inc. *Solaris Open Windows: The ToolTalk Service*, 1991.

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

[25] G. Valetto and G. Kaiser. Enveloping "Persistent" Tools for a Process-Centred Environment. In W. Schäfer, 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, 1995.