PRIME:
Towards Process-Integrated Environments[1]

Klaus Pohl, Klaus Weidenhaupt, Ralf Dömges,
Peter Haumer, Matthias Jark, Ralf Klamma

Lehrstuhl Informatik V (Information Systems)

RWTH Aachen

Ahornstr. 55, D-52056 Aachen, Germany

To appear in: ACM Transactions on Software Engineering and Methodology, 1999

[1] The PRIME framework is a result of the Project 445/5-1 "Prozeßintegration von Modellierungsarbeitsplätzen" funded by the Deutsche Forschungsgemeinschaft. The PRIME framework has been used to implement two prototypical process-integrated environments within the ESPRIT Reactive Long Term Research 21.903 Project CREWS (funded by the European Community) and the Collaborative Research Center (SFB) 476 IMPROVE (funded by the Deutsche Forschungsgemeinschaft).

Abstract:

Research in process-centered environments (PCEs) has focused on project management support and has neglected method guidance for the engineers performing the (software) engineering process. It has been dominated by the search for suitable process modeling languages and enactment mechanisms. The consequences of the process orientation on the computer-based engineering environments, i.e. the interactive tools used during process performance, have been studied much less. In this paper, we present the PRIME (PRocess-Integrated Modeling Environments) framework which empowers method guidance through process-integrated tools. In contrast to the tools of PCEs, the process-integrated tools of PRIME adjust their behavior according to the current process situation and the method definitions. Process-integration of PRIME tools is achieved through

- the definition of tool models;
- the integration of the tool models and the method definitions;
- the interpretation of the integrated environment model by the tools, the process-aware control integration mechanism, and the enactment mechanism;
- the synchronization of the tools and the enactment mechanism based on a comprehensive interaction protocol.

We sketch the implementation of PRIME as reusable implementation framework which facilitates the realization of process-integrated tools as well as the process-integration of legacy tools. We define a six-step procedure for building a PRIME-based process-integrated environment (PIE) and illustrate how PRIME facilitates change integration on an easy-to-adapt modeling level. Following the six-step procedure we have implemented two process-integrated environments (PRIME-CREWS and TECHMOD) which have been applied in small case studies.

Categories:

D.2.1 [Software Engineering]: Tools
D.2.2 [Software Engineering]: Computer-aided software engineering (CASE)
D.2.6 [Software Engineering]: Integrated environments
D.2.6 [Software Engineering]: Interactive environments
D.2.11 [Software Engineering]: Software Architectures
D.2.13 [Software Engineering]: Reusable Software
D.3.3 [Programming Languages]: Frameworks
H.4.1 [Information Systems Applications]: Workflow Management
J.6 [Computer aided engineering]
K.6.3 [Software Management]: Software process

Keywords: process-centered environments, process-integrated environments, method guidance, process modelling, process-sensitive tools, tool modelling, tool integration, PRIME

1 Introduction

1.1 Process-Centered Environments

During the last decade a tendency of moving from product-oriented computer supported development environments to process-oriented environments, so-called process-centered environments (PCEs), could be observed. The process improvement paradigm popularized by approaches such as the SEI Capability Maturity Model [1] or Total Quality Management [2] stresses the necessity to focus on the production process in order to achieve better product quality at decreased costs. Hence, product-oriented environment support in form of data integration mechanisms (such as repositories or standards for data exchange formats) have to be complemented by process-oriented support functionality. Striving for continuous process improvement implies that process knowledge is never stable. Accumulated experiences from former projects may indicate better ways for performing certain parts of the process. Moreover, each project imposes its own specific restrictions on how development processes should be carried out.

The explicit definition of processes in PCEs is a prerequisite for an easy adaptation of the development processes to project specific needs and the integration of process changes. In contrast, the process support offered by product-oriented environments is hard-coded. There exists no explicit process definition. Process changes require reprogramming, and are thus hard to accomplish.

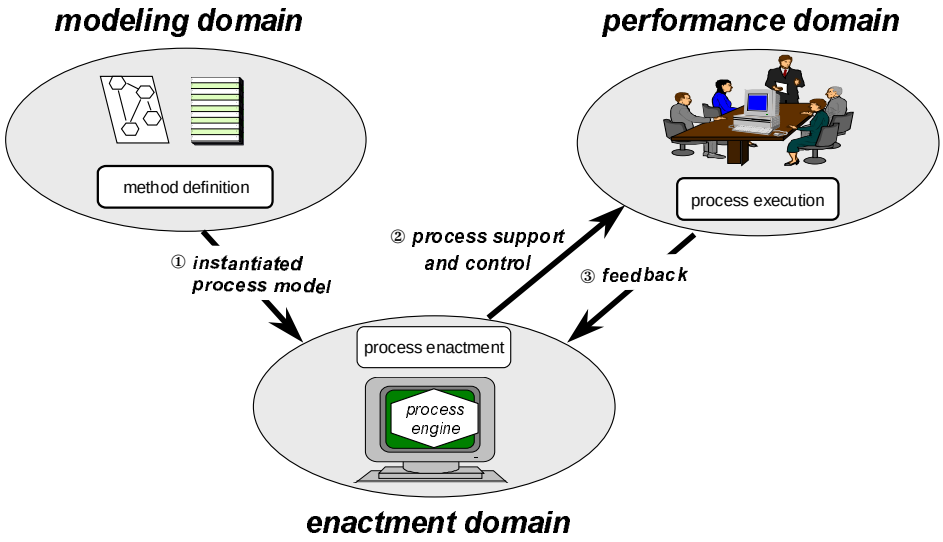

Figure 1: Three domains of software process support.

PCEs comprise three conceptually distinguishable domains [3], [4]: the modeling, the performance, and the enactment domains (Figure 1). The *modeling domain* comprises all activities for defining and maintaining process models using a formal language with an underlying operational

semantic which enables mechanical interpretation of the models. The *enactment domain* encompasses what takes place in a PCE to support (guide, enforce, control) process performance; this is essentially a mechanical interpretation of the process models by a so-called process engine. The *performance domain* is defined as the set of actual activities conducted by human agents and non-human agents (computers).

Process support provided by PCEs can be characterized by the typical interactions between the three domains (Figure 1):

1. A process model is instantiated by binding process parameters such as resources and time scheduling to project specific values, and then passed to the enactment domain;
2. Based on the interpretation of the instantiated model, the enactment domain supports, controls, and monitors the activities of the performance domain;
3. The performance domain provides feedback information on current process performance to the enactment domain. This is a prerequisite for adapting process model enactment to the actual process performance and enabling branches, backtracks, and loops in process model enactment.

Research in the PCE area has concentrated on the modeling and enactment domains [5]. It has resulted in a set of mature process modeling languages and enactment mechanisms, e.g. [6], [7], [8], [9], [10], [11], [12], [13], [5], [14]. Excellent overviews on the research in the PCE area and comparisons of different approaches can be found in, e.g., [15], [16], [17], [18], [19], [4].

Unfortunately, the consequences for the interactive tools of the environment have been studied much less. The tools of PCEs do not provide integrated, definition-conform stakeholder guidance.

1.2 Project Management Support vs. Method Guidance

Existing PCEs focus mostly on *process (project) management support*. Method guidance for the engineers who are actually performing the (software) engineering process has largely been neglected. Process management support offered by existing PCEs and method guidance for the engineers performing the process differ in two main aspects.

- The primary concern of project management support provided by PCEs is to *coordinate the interworking of people* at the task level, for example, to ensure that tasks (reviewing a design document, testing a module etc.) are performed in a certain order, information flows correctly, or time constraints are met. In contrast, method guidance aims at supporting the engineers performing these tasks [20], [21] such as to guide the engineer during the refinement of an entity type and the required adjustment of the data flow diagrams, or to enforce the recording and use of traceability information[22];
- The artifacts managed in current PCEs and the artifacts considered by method guidance vary significantly in their granularity. While most PCEs consider the products under development

and their relations at the *level of documents* (such as Entity Relationship diagrams, design documents, or test modules) method guidance requires the consideration of a more *fine-grained product structure* like entities, relationships, attributes.

Thus, *method guidance aims at supporting the stakeholder performing a task* based on the explicit definition of best practice gained by generalizing from (individual) experiences. Whereas process support is defined in so called process models/definitions, method guidance is specified in method models/definitions. In well understood domains, method guidance could be very restrictive. For example, the handling of an insurance claim and the criteria for accepting or rejecting the claim could be precisely defined. The tools used to process the claims should thus enforce and support the employees in meeting the actual claim handling definitions. Such kind of support is requested by and researched in the workflow community (cf., e.g., [23]).

In the case of more creative tasks such as design activities, a complete and strict definition of the task is not possible. But also in creative processes certain sub tasks must be performed in pre-defined situations, and/or the performance of some sub tasks can be restricted by some predefined criteria. Those subtasks can be predefined and used to guide the process execution. For example, it might be defined that whenever a change request requires an adaptation of an approved document, the change request has first to be approved following a predefined procedure and each change in the document has to be reported to all stakeholders influenced by the change. Process execution should thus in general be unrestricted, but support the execution of predefined procedures (sub tasks) whenever possible. Examples for areas in which such kind of support is required are method-conform development of conceptual models [20], [21], ensuring consistency between design documents [24], managing inconsistencies during system development [25], or constructing and simulating flowsheet models for chemical plants [26].

1.3 Providing Integrated Method Guidance

A prerequisite for method guidance is that there are efficient ways to communicate the project-specific method definitions to the stakeholders and, as far as possible, ensure that the stakeholders actually apply the definitions during process execution. There are three principal solutions to achieve this: *handbooks, separate guidance tools*, and *integrated guidance.*

Handbooks: This kind of support is widespread in current industrial practice, where guidelines and instructions for performing a process or task are provided in handbooks. The usual way of orchestrating method support in the handbook and the actual process performance is: during a learning phase each stakeholder learns the guidelines, instructions etc. If the humans performing the process/task do not remember the guidelines in the corresponding situations, the method support will not be used. In other words, the actors must know what are "legal" or "good" steps according to the handbook and under which circumstances these steps can be applied. Of course, the stakeholder can use the handbook as reference manual to look for details concerning the method definitions (support), but they have to know (and remember) when to look and for what.

Changes in the project-specific definitions cause obvious problems: each potential actor must be informed about the change and it must be ensured, e.g. by training, that everybody adapts his/her knowledge according to the change. Using handbooks for providing method guidance also hinders method improvement. The method definitions can be of high quality, but can be neglected during task performance. If the method definitions are neglected, the recorded trace and monitoring data can wrongly indicate the need for a method improvement.

Separate Guidance Tools: In contrast to handbooks, a separate guidance tool offers direct support for the process performer. It presents the project-specific method definitions applicable in a given situation to the stakeholders. Such a tool can range from a web-based browser providing selective access to the documented method guidance to a task manager which guides, e.g., the engineer by notifying him about the task to be performed next. Moreover, the guidance tool can be empowered to invoke the execution of an action or tool, to set up a special working environment, or to remind the stakeholder about existing definitions. If project-specific adaptations of the method definitions are integrated in the guidance tool, then the new definitions are immediately used during process execution by all actors. Furthermore, the performance of a task can be better controlled, monitored and traced. Thus, compared with a handbook, a guidance tool offers significantly better support.

Nevertheless, providing project-specific method guidance in a separate tool has some shortcomings such as:

- *The number of user interfaces increases*. The user has to interact with the tools used to perform the tasks and with the guidance tool to report the status of task performance. This is especially difficult if detailed feedback information is required.
- *Process performance and guidance are not integrated.* The task performer is responsible for keeping the guidance tool up to date, e.g. reporting the execution of an action, the results, the actual state, etc. It can thus not be ensured that the current task status of the task performance is known in the guidance tool.
- *Guidance is typically coarse grained*. The separate guidance tool has only limited knowledge about the actual process state which is mainly restricted to abstract document and task states. In contrast to the interactive engineering tools a separate guidance tool does not know the states of each product component nor the actions actually being performed. As a consequence, the guidance offered is typically at a more abstract level (coarse grained).
- *The actor can execute another action*, but report the execution of the action suggested by the guidance tool. Thus the feedback information can simply be wrong, imprecise or idealized to meet certain expectations. Capturing wrong traceability information can lead to unintended method "improvement" (it can even spoil the method definitions).

Separate guidance tools such as task managers are typically being used in PCEs to inform the user about the enactment state and to obtain feedback information about the execution of a task.

Integrated Method Guidance: The limitation of a separate guidance tool can be avoided if the project-specific method guidance is integrated into the interactive tools used to perform the activities or tasks, e.g. a tool for handling insurance claims, a CAD tool or a UML editor. Integrating method guidance into the interactive tools means that the interactive tools

- inform the user about the actual method definitions applicable for the current situation. This includes to notify the users about existing method guidance for performing the activity and/or task at hand;
- guide the user in choosing among defined alternative ways of performing an activity or task;
- remind the user if the actual task performance leads to a violation of the best practice definitions, e.g. to data inconsistencies;
- restrict the services provided according to the method definitions. For example, if the method definition does not allow a specialization of an entity in certain situations, the tool should disable the functionality.

The advantages gained by providing integrated method guidance are:

- The stakeholders need not necessarily be aware of the project-specific method definitions since their interactive tools act as on-line assistant during process performance.
- Updates of the method guidance are directly available in the tool environment of each workplace.
- Less training effort on “best practices” is required. Integrated method guidance ensures that the "best practice" is actually being applied, and thus products of higher quality are produced, errors are avoided and expensive rework is reduced.

Integrated method guidance should, however, not be too restrictive. The user should always be able to neglect the method guidance offered and to abort the method definition being enacted.

Technically, integrated method guidance can be achieved by either hard-coding the guidance in the tool environments or by providing the guidance based on the interpretation of explicit method definitions. Hard-coding the method guidance might be well suited for domains in which the defined best practice (method guidance) is stable. If there are frequent changes in the method definitions (like in creative processes) method guidance should, similar to process guidance in PCEs, be based on the interpretation of explicit method definitions.

To empower the tools in the performance domain to offer model-based, integrated method guidance and to provide detailed feedback information according to the method definition, a better integration of the engineering tools and the enactment mechanism is required. Such an integration is also essential for achieving a synchronization of the enactment and performance domains. The synchronization is required to adjust the support offered to the current performance state and to provide sufficient feedback information. The need for a tighter integration of the enactment and the

performance domains has been widely recognized, e.g. [27], [3], [28], [29], [30], [31], [32],[33], [5], [34], [4], [35], [36], [37], but no systematic approach has been proposed so far.

1.4 Structure of this Paper

To achieve model-based, integrated method guidance, we argue in this paper that the interactive engineering tools of the computer-based environment must be process-integrated. In Section 2 we define a set of requirements for *process-integrated environments* (PIEs).

Towards an ideal solution for achieving these requirements we sketch the basic ideas underlying our PRIME framework (Section 3). We show that process-integration of tools can be achieved through

- the definition of the tool capabilities in tool models, the integration of the tool models with the method definitions and the interpretation of the so gained integrated *environment model* during process execution (Section 4);
- an integration of the enactment and performance domains by an *interaction protocol* and a *process aware control integration* mechanism which controls message distribution based on the interpretation of the integrated environment model (Section 5).

Based on these integration ideas, we derive an architecture for process-integrated modeling environments, called *PRIME* (Section 6). The generic architectural components of PRIME have been implemented as an object-oriented implementation framework.

To facilitate the process-integration of legacy tools we outline an extension of the generic tool architecture of the PRIME implementation framework and elaborate on a set of APIs (application programming interfaces) to be provided by a legacy tool to be fully process integrable. Using those extension we sketch the process-integration of VISIO, a commercial CAD tool (Section 7).

For building a PRIME-based process-integrated environment (PIE) we outline a six-step procedure (Section 8) and demonstrate how PRIME supports the integration of changes on an easy-to-adapt modeling level (Section 9). Finally, we summarize the main contributions of the PRIME framework and provide an outlook on future work (Section 10).

PRIME has been used to implement two prototypical process-integrated environments, PRIME-CREWS and TECHMOD. Throughout the paper we refer to the two environments for illustrating certain aspects.

2 Requirements for Process-Integrated Environments

In the enactment domain, method definitions[2] are enacted to drive the modeling process. In the performance domain, humans use (interactive) tools to execute the proposed method or process steps. Providing integrated method guidance for the engineers requires an integration of the enactment and performance domain. Such an integration mainly has to cope with aspects of data, control and process integration. While we share Wasserman's view on data and control integration [38] (see also [39]), we claim that process integration requires certain features which are not discussed in literature so far.

The requirements elaborated in the next subsections are thus significantly more comprehensive than those discussed, for example, by [40], [3], [28], [30], [31], [37],[41], [32] whose analysis is mainly based on the weak integration of the enactment and performance domains in existing PCEs. They slightly extend the requirements discussed in [42].

2.1 Data Integration

The process engine must pass data to the tools of the performance domain such as parameters of a service request. The tools return feedback information to the enactment domain, for example, the results obtained from executing the requested service.

Both kinds of data exchange require an agreement about the information to be exchanged. The data to be exchanged can be coarse-grained or fine-grained depending on the granularity of the method definition. Roughly speaking, data integration can be achieved by defining a common data base schema and recording the data in a logically centralized database (such as PCTE [43], [44]), and/or by agreeing on the data and their format for each message type (e.g., CDIF [45], XMI/SMIF [46]). Whereas the second kind of integration requires that the data is actually included in each message, the first one enables the exchange of object identifiers or views on the common data base which is especially convenient and efficient for exchanging large amounts of data.

2.2 Service Integration

A *tool service* is a functionality provided by a tool which can be called from outside such as the creation of a certain artifact, the compilation of source code, or printing a document. Tool services can vary in their complexity. To ensure that the tools of the performance domain can execute the services requested from the enactment mechanism, the tools must be considered when defining method definitions.

---

[2] For the process-integration requirements discussed in this sections, the terms method definitions/models can be seen as equivalent to process definitions/models.

Therefore, the method engineer has to collect information about the available tools, their services and the service invocation such as parameters required from various sources such as manuals, program documentation, personal knowledge and/or experience. Considering the heterogeneous environments and work settings which exist today in industry, mechanisms are required which systematically support the method engineer in finding and assigning adequate tool support to certain method steps. If the capabilities of the tools like their services and the in and out parameters of the services are defined at a conceptual level the method engineer can be supported in relating the tool services to the method definitions. For example, the tool and method definitions can be compared and discrepancies, such as lack of sufficient tool functionality or wrong assignments, can be detected.

Current process modeling formalisms lack comprehensive modeling concepts for representing tool resources at the same conceptual level as processes. They offer only limited, low-level constructs for representing service invocation. Examples include the black transitions in SPADE [7], or the wrapping techniques for the black-box integration employed in the OZ environment [32], [36].

2.3 Invocation of Method Fragments

Methodical support for creative processes cannot be fully predefined. Many criteria which influence the actual performance are not known a priori and some method steps themselves are poorly understood. The actual method execution is thus often driven by humans who, depending on the given situation, decide what to do next. Method execution depends on intelligent and creative individuals who make the right decisions. It is thus important that the computer-based environment does not restrict the humans in their creativity.

On the other hand, even in creative processes there exist steps which are well understood, do not depend on unknown criteria and, thus, can be predefined and must be followed [47], [4]. Examples for such "steps" are the integration of change requests of formally approved documents, documentation and traceability guidelines imposed by a contract, or the assessment of the creditability of a customer. To increase the productivity and the quality of the product under development, such method knowledge should be used, whenever possible, to guide the engineer. As a consequence, the understood steps should be defined in, what we call, *method fragments*.

In contrast to a method definition which typically defines a whole method, a method fragment is a partial definition which specifies the guidance for a well understood method part. Nevertheless, a method fragment can and should be used to guide the users of a computer-based environment whenever the current situation demands/indicates the execution of the method fragment.

The computer-based environment should, of course, support the user in the invocation of a predefined method fragment. Process-integrated tools must thus provide means for initiating the execution of predefined method fragments. This can be achieved by comparing the current process situation with the method fragment definitions.

2.4 Process Sensitive Tools (Informing the User about the Enactment State)

A tight integration between the enactment and performance domains can only be achieved if both domains consider the process status of each other. The enactment domain has to consider the current performance state for deducing the steps to be performed next (Section 2.5), whereas the current enactment state has to be reflected in the performance domain.

Only if the user is aware of the enactment state and the method definitions she or he is able to understand the guidance provided by the enactment domain. Moreover informing the user about the current enactment state (i.e. the current performance state assumed by the enactment domain) empowers the user to correct wrong and change undesired states. In existing PCEs the enactment state is, if at all, typically accessible for the user via a separate user interface (tool). As a consequence, the actual state is often not considered when interacting with the computer-based engineering tools. To ensure that the user is aware of the current enactment state, we argue that the interactive tools used should be process sensitive.

A process sensitive tool adapts its behavior (the user interactions allowed and the services provided) according to the current enactment state and the method definitions. For example, the selectability of product parts may be restricted to the ones allowed in the current state, or the product parts on which a service can be performed might be highlighted to draw user attention to them. Moreover, a process sensitive tool empowers the user to activate predefined method fragments. Since method fragment definitions are subject to frequent change, the activation of predefined fragments should not be hard-coded in the tool. Instead, the activation should be based on the actual method fragment definitions. In other words, the tools have to be process-aware and have to know the activation criteria for the defined method fragments.

Supplying the performance domain with knowledge about the enactment state is straightforward in situations where a particular service has to be performed on a specific product part. In this case, the relevant product parts are passed as parameters of the service request. For example, if the entity "book" should be deleted, the entity "book" is passed as parameter of the delete service.

If there are alternatives among which the user can choose passing the required information to the performance domain is much more complicated. The enactment domain must inform the tools about the alternative services (strategies) allowed and the product parts on which the services can be applied. For example, assume that for integrating a certain type of requirement, the user can choose between two alternative services, namely the definition of a new entity and the refinement of an existing entity. To guide the user, the tool must display the allowed alternative services together with the existing ER-diagram. Moreover, the tool should only enable the selection of entities since the two services can only be performed on entities. Attributes, relationships, role names and cardinalities should thus not be selectable.

2.5 Feedback Information (Informing the Process Engine about Performance State)

For adjusting the enactment state according to the actual process execution, the performance domain must provide feedback information about service executions. The data to be exchanged depends on the service executed. Consequently, the feedback data has to be defined as out-parameters for each service type (see service integration above). In addition, information about the current performance state including unforeseeable events such as a process deviation have to be provided. This information can either be created by observing (monitoring) activities, or directly provided by the user.

Technically, a control integration mechanism should be used to take care of correct distribution of the feedback information.

The problem of gathering feedback information from the performance domain has been widely recognized. For example, SPADE [7] has introduced a specific Petri-Net construct, the user input place. Message events generated by the tools have to be reified into tokens of such places. In Provence [48], [49] the enactment mechanism captures events from the performance domain via a monitoring system for operating system traps such as file system accesses. But mapping performance domain events to feedback information understood by the enactment mechanism is by far not trivial, e.g. deducing that saving a file in a text editor means that a bug fix in the source file has been completed.

2.6 Synchronization of Enactment and Performance Domains

The definition of an interaction protocol and its consideration within each domain is a prerequisite for synchronizing the states of both domains. In current PCEs, the interaction between the enactment domain and the tools is typically established by an implicit client-server relationship: The enactment domain acts as a client which requests the execution of a tool service. Conversely, the tool plays the role of a server which executes the service and returns the results (feedback information) to the enactment mechanism. This simple cooperation pattern is sufficient as long as we consider traditional tools which are not process-integrated.

The more active role of process-integrated tools (Section 2.3 - 2.5) requires an interaction protocol between the two domains more elaborated than the client-server. Such a protocol should, for example, distinguish between different process states such as normal process performance, process deviations, the performance of automated services, or user choices.

2.7 Process-Aware Control Integration Mechanism

In contrast to service integration, which considers the service interfaces, a control integration mechanism is required for transmitting particular service requests and feedback information

between the components of a process-integrated environment. A control integration mechanism is responsible for passing the requested service to a tool which is able to execute the service. To enable correct physical distribution of the service requests and the feedback information provided after service execution, the control integration mechanism has to be aware of the services provided by a particular tool. In addition, service and feedback distribution have to consider relevant knowledge defined in the method model. For example, when distributing a service the control integration mechanism has to consider if the model restricts the allocation of possible resources needed for performing the service, or if the model explicitly defines a particular service provider. Thus, either the control mechanism must be process-aware, i.e. it must know the relevant parts of the actual method definition, or the enactment domain must instruct service distribution according to the method definition. Most existing PCEs (e.g. SPADE [50], MELMAC [11], Merlin [51], Process WEAVER [12]) offer neither a process-awareness of the control mechanism nor the ability of the enactment domain to control service distribution.

Existing control integration mechanism like FIELD [52], BMS of HP's Softbench [53], ToolTalk [54], CORBA [55] or (D)COM [56] provide an excellent foundation for implementing a process-aware control integration mechanism.

3 PRIME: Key Solution Ideas

Our PRIME framework provides solutions to the seven requirements discussed in Section 2. PRIME is based on three basic ideas:

1. The explicit definition of *tool models* and their integration with method definitions (Section 4). We argue that, from a modeling perspective, tools should no longer be treated as second class citizens. Instead, the capabilities of tools should be explicitly defined and related with the process/method definitions. The integration of the tool and the method definitions forms an *environment model* which lays the foundation for the process-integration of the interactive engineering tools and for a tighter integration of the performance and enactment domains.
2. The *integration of the performance and the enactment domains* (Section 5). The synchronization of both domains is achieved by a comprehensive *interaction protocol* which defines the principal behavior of both domains. The interaction protocol defines a richer interaction pattern which, in contrast to most PCEs, empowers both domains to act as a client. The enactment domain can request the execution of tool services and the performance domain can request the enactment of predefined method fragments. Model conform distribution of service requests and feedback information is guaranteed by the *process-aware control integration* mechanism which controls message distribution based on the interpretation of the environment model.
3. *Generic architectures* for process-integrated tools and enactment mechanisms which, together with a process-aware control integration mechanism, comprise the PRIME

implementation framework (Section 6). Both architectures ensure that method execution is in accordance with the environment model and the interaction protocol. The generic tool architecture facilitates process sensitivity by supporting the invocation of predefined method fragments and guaranteeing that the guidance provided to the user corresponds to the method definition and the current process situation. The generic enactment architecture handles enactment requests of the performance domain by enacting the requested method fragments and provides means for an easy integration of existing enactment mechanisms.

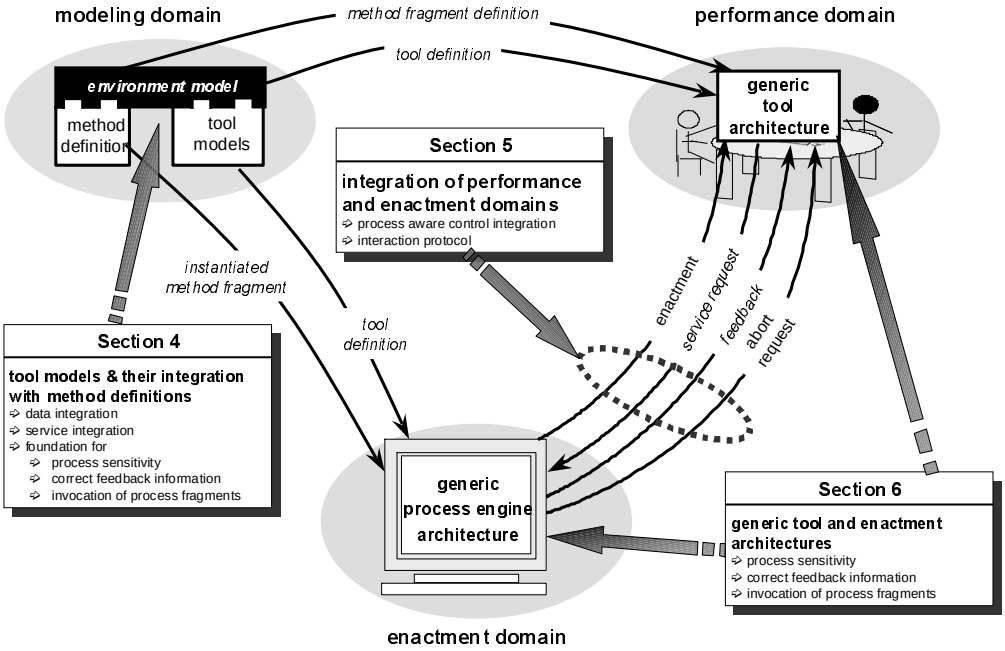

Figure 2: The three key solution ideas of PRIME. <pohl2.ps, pohl2.tif >

Comparison with Related Work

Whereas the main focus of PRIME is to establish integrated, model-based method guidance based on a tight integration between the enactment and the performance domain (i.e. the process engine and the interactive tools used to perform the process), most research contributions do not consider the integration of tools in PCEs although the problems of a posteriori integration of existing CASE tools have been widely recognized (e.g.[19], [34], [57], [31]). Bandinelli et al. argue that

> *"a posteriori tool integration (e.g. by means of wrappers) could be less effective since a tool is still seen as a monolithic 'operator' " [50].*

Consequently, existing PCEs do not offer process-integrated tools.

An exception, where means for white-box integration are offered, is the GTSL approach [29] developed within the GOODSTEP project [58] which aims at the *generation* of specific tool services, schemata, and consistency checks from tool specifications which are coupled with process

models. GTSL mainly provides solutions to the service and data integration problem, but does not provide means for the invocation of method fragments or for dynamic adaptation of the tool behavior according to the process definition and the enactment state.

Meta-CASE environments like MetaEdit+ [59] are based on the generation of tools according to a specification. They focus mainly on notational aspects, but lack process-orientation (see [60] for a detailed comparison of the complementary adaptability mechanism provided by MetaEdit+ and PRIME).

Existing process modeling languages focus on the constructs and their semantics needed to define processes and enactment mechanisms which can be used to interpret the definitions. They mainly neglect tool definitions. Some provide low-level constructs for the invocation of foreign programs like black transitions in SLANG [7], or the binding of abstract process operators to tools during process instantiation in ALF [61].

Control-oriented tool integration approaches like FIELD [52] and its commercial derivatives (such as HP's BMS [53] and Sun's ToolTalk [54]) as well as object-oriented distribution infrastructures like CORBA [55] or (D)COM [56] store tool (service) descriptions in the interface repositories accessed by the message servers/object brokers. However, they provide limited means for defining processes.

As a consequence, if at all, tool and process models coexist in the message server repository and in the process repository without a systematic approach for assuring consistency. Many PCEs like SPADE [7], Process WEAVER [12], EPOS [62], and Merlin [51] employ such mechanisms for invoking tool services, although the tool models used by the message server and the process models used by the enactment domain are not (systematically) integrated.

Tool invocation in existing PCEs is mostly restricted to the invocation of "atomic" tool actions or services. User guidance by adapting the accessible objects and operations through guidance services is not (systematically) supported. The interactive tools of existing environments are not process-integrated.

In the Multi-Tool-Protocol (MTP) approach [36] the single/multi-user and single/multi-task capabilities of tools are explicitly defined. The explicit tool classification empowers MTP to provide better tool invocation support than conventional black box approaches. In contrast to PRIME, MTP does not provide any means for facilitating the adaptation of tools according the process definition and the actual process situation (enactment state). Nevertheless, the actual tool invocation in a PRIME based environment could make use of the MTP facilities.

The FIELD-based Forest environment [63] is an attempt to establish a central description of processes and tools. Forest extends the tool-related message distribution patterns stored in the message server by so-called policy descriptions which can be regarded as primitive process definitions. Although this approach improves the integration of tool and process models it provides no systematic means for establishing process-sensitive tools. It does also not support the invocation

of method fragments, and thus, similarly to most other PCEs, it does not allow a more active role of the humans executing the process which is one of the main achievements of PRIME.

Especially in desktop environments (Windows, MacOS), the document-centered paradigm has become more and more popular in recent years. Document models such as OLE [56] or OpenDoc [64] provide the technical foundation for blurring the boundaries between individual tools in that certain functionality is (from a presentation perspective) no longer bound to specific tools but to document objects. Document objects can be nested within container documents forming so-called compound documents. The tool functionality is then presented to the user, e.g., through context-sensitive menus which display only those functions which are applicable in the selected document context. In this sense, the "document acts as an intelligent assistant of its user" [65] and the individual tools providing the operations on the documents step aside from a user perspective. However, the context sensitivity in the document-centered paradigm mainly deals with the functions which are bound to the currently activated document context. It is not coupled to any method or process definitions. Functionality across documents defined in method or process definition is thus only rudimentarily supported. Moreover, there is no generic, model-based mechanism for defining functionality across documents nor for attaching the invocation of method fragments to certain document parts or even product constellations across different documents.

In summary, the need for tighter (process) integration of engineering tools was recognized and some partial solutions to the problem exist. So far, no comprehensive approach was proposed which establishes process-integration of tools and, in addition, enables the humans performing the process to play a more active role.

4 Integrated Tool and Process Models

According to the requirements discussed in Section 2, there are *three types of services* in process-integrated environments (PIE): automated, guidance, and enactment services [42].

Automated services require no user interactions and are executed by the tool according to the service request obtained by the enactment domain. An example for an automated service is the compilation of source code or the automated recording of traceability information.

Guidance services guide the user in making a selection among a set of alternative services and/or product parts. If the execution of a *guidance service* is requested, the tool must adapt its behavior (the services offered and the product parts displayed at its user interface) according to the method definition and the information obtained with the service request. An example for a guidance service is the refinement of an entity type which defines two alternatives: introducing a discriminating attribute or specializing (subtyping) the entity type. The tool has to display the defined alternatives to the user (e.g. as menu options), and the user has to choose the alternative to be executed.

Enactment services enable the tools to request the enactment of a complex method fragment from the enactment domain. An example for a complex method fragment is the "subtyping" of the entity. This fragment consists of a set of steps (services) which have to be performed in a certain order. Thus, the fragment has to be enacted by the process engine. If the user chooses the subtype alternative the ER editor has thus to request the execution of the complex method fragment by the process engine.

Since a method model defines when and how a service (method fragment) should be performed, the process meta model (process modeling language) must provide appropriate concepts to define the three service types as well as their situated invocation (Section 4.1).

The tool model defines the services provided by a tool. A tool meta model (tool modeling language) must thus provide appropriate concepts to define the capabilities of the interactive tools used to perform the services (Section 4.2).

In Section 4.3 we describe the integration of the tool and process/method meta models into the *environment meta model*. In Section 4.4 we summarize the contribution of the environment meta model to the process-integration requirements outlined in Section 2.

4.1 The Process Meta Model: Defining Method Fragments

To define the three service types we suggest to use the contextual process meta model developed in the ESPRIT project NATURE (see [66], [67], [68], [47], [4] for a detailed description). Figure 3 introduces the key concepts of the meta model and their relationships using the OMT notation [69].

Briefly, a *situation* is built from product parts of the *product* undergoing the development process. An *intention* reflects the goal to be achieved in a given situation. A *context* represents a meaningful relation between a situation and an intention. Thus, the meta model provides concepts for the explicit representation of situations and the goals to be achieved in such situations. The notion of context is further refined into executable, choice and plan contexts:

- *Executable contexts* represent the part of the definitions which can be strictly enforced, or even automated. An executable context is operationalized by performing the *action* related to this context. Performing the action changes the product and may thus generate new situations;
- *Choice contexts* represent the part of the definitions in which the user has to make a decision. For each choice context, at least two *alternative*s must be defined. An alternative can be another choice, executable, or plan context. For each alternative, *argument*s (pros and/or cons) can be provided to guide the application engineer in choosing one of the alternatives;
- *Plan contexts* define a strategy to be followed to fulfill a particular intention (goal). A plan context defines a certain order on a subset of arbitrary contexts. It can be used to enforce the

application engineer to deal with the contexts in the order defined. It thus corresponds to an enactment service provided by the process engine.

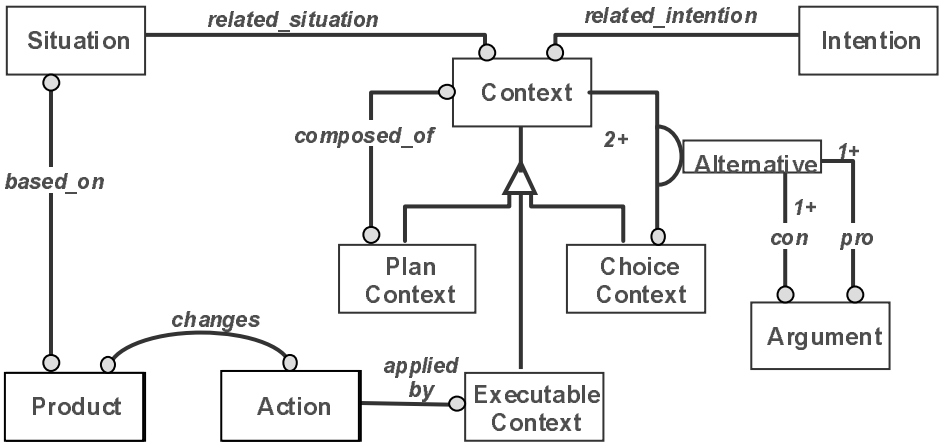

Figure 3: The NATURE process meta model [47], [66].

4.1.1 Defining Method Fragments Using NATURE's Process Meta Model

To define method guidance, the process meta model has to be instantiated. Thereby a concrete method definition is established. The definition of automated and guidance services is straightforward.

Automated services are defined as executable contexts. For each executable context, the situation and the intention which together specify the activation criteria have to be defined. Moreover, each executable context has to be related to the action to be performed whenever the executable context is activated.

Guidance services are defined as choice contexts. Similarly to executable contexts, also for each choice context a situation and an intention has to be defined. Moreover, for each choice context the allowed alternatives have to be defined. In addition, for each alternative arguments supporting or declining their choice can be defined.

Enactment services are defined as plan contexts. Whereas the definition of the criteria for activating a plan context (situation and intention) is similar to the definition of executable and choice contexts, the definition of the plan itself is more complicated. According to the process meta model a plan context is composed of a set of choice, executable, and/or other plan contexts. In addition, the sequence of activation of those contexts has to be defined by some kind of *control flow*. Thus, the definition of the plan context requires a language with higher expressiveness. For defining the control flow of plan contexts, we do not argue to extend the contextual meta model with additional concepts and an operational semantics. Rather, we suggest to represent the concepts of our model in

an existing process modeling language which supports the definition of control flows and has an operational semantics. The main requirements for such a language are enactability, modularity for enabling the invocation of fine-grained method fragments, and composability for nesting method fragments (arbitrary contexts). If a particular language has been chosen, the three context types and the notion of situations and intentions must be represented by concepts provided by the language.

4.1.2 Service Definition in SLANG

For our prototypical environments (see Section 8.2) we have chosen the Petri-Net language SLANG [7] and the imperative language C++. In both languages we have defined templates which support the definition of the three context types. In the following, we sketch the context definition in SLANG.

Plan contexts and choice contexts are modeled as sub-nets in SLANG. Executable contexts are modeled as transitions (depicted as gray bars; see Figure 4). The situation type and the intention of each context is mapped within a SLANG net to a set of situation places (depicted as circles enclosing a square) and an intention place (depicted as circles enclosing a triangle). Each situation place carries structured tokens representing the product parts of the corresponding situation type. Thus, for each transition and for each sub-net representing a context, a set of "entry" places is defined. Similarly, "output" places are used to define the result of the execution of a context. In the case of executable and plan contexts, the output consists of a set of places representing the created/changed product parts. The result of a choice context is defined by a set of situation/intention place pairs which represent the possible alternatives of the choice context.

To activate a context defined in a SLANG (sub)net, the required tokens must be moved into the corresponding situation and intention places, e.g. by mapping the output places of a preceding context to the "entry" places of another context and by filling its intention place through additional transitions. We call such additional transitions control transitions (see [70] for details).

4.1.3 Defining Method Guidance: An Example

To illustrate the definition of a plan context in SLANG we specify the method guidance for subtyping an entity type and adjusting the corresponding data flow diagrams.

The subtyping of an entity type is likely to influence the definition of the data flow diagrams (DFDs). For example, the entity being subtyped could correspond to a data store defined in a DFD. In some cases it is useful to refine the effected data stores and/or the adjacent data flows or even to create new processes which operate differently on the refined DFD elements. The guidance is thus defined as plan context *PC_SubtypeEntityAndAdjustDFD*.

Figure 4 depicts the specification of this plan context in SLANG. Within the plan context *PC_SubtypeEntityAndAdjustDFD* the subtyping of an entity is defined by the plan context

PC_SubtypeEntity (see upper left part of Figure 4). This context is activated by the start transition which maps the tokens of the situation place *EntityToBeSubtyped* to the situation place *SuperEntity* and fills the intention place *SubtypeEntity*. The data stores and adjacent flows related to the subtyped entity are retrieved by the executable context *EC_GetDependentObjects*. The situation places of this context consist of the source object (pre-filled with the entity type to be subtyped) and the target types of the dependent objects to be retrieved (pre-filled with a token of the type *DFD_Element*). A token is moved into the intention place of the executable context *EC_GetDependentObjects* after the plan context *PC_SubtypeEntity* has been executed. The intention place *GetDependentObject* is defined as output place of the plan context (cf. Figure 4). This activates the executable context *EC_GetDependentObject* which retrieves the dependent objects. Those objects are passed to the output place *DFD_Elements*.

According to the definition of the plan context *PC_SubtypeEntityAndAdjustDFD,* the user can choose which DFD element is going to be adapted first, i.e. the place *DFD_Elements* is defined as situation place of the choice context *CC_Adapt_DFD_Element* and the intention place of this context is filled after the executable context *EC_GetDependentObject* has been terminated. In addition to the DFD element, the user has to choose one out of the four alternatives defined for this choice context (middle part of Figure 4):

- *Adapt the selected data flow element*: Method guidance for the adaptation of the DFD element is defined as plan context (*PC_AdaptDFDElement*). Among others, the plan context contains a choice context offering three different alternatives to the requirements engineer, namely to specialize an adjacent data flow of the data store which corresponds to the specialized entity, to specialize the data store itself, or to partition the data store and introduce new processes and flows which operate differently on the data stores;
- *Add to the task* This alternative is defined as executable context *EC_AddToTaskList* which adds the adaptation of the selected DFD element to the open task list of the requirements engineer;
- *Leave DFD element unchanged*: The executable context *NoChangeRequired* removes the token representing the selected DFD element from the place *DFD_Elements*, i.e. by choosing this alternative the requirements engineer indicates that the selected DFD element (or elements) need not to be changed;
- *Quit adaptation*: If the engineer chooses this alternative she or he indicates that the remaining DFD Elements need not to be adapted and thus the plan context *PC_SubtypeEntityAndAdjustDFD* terminates.

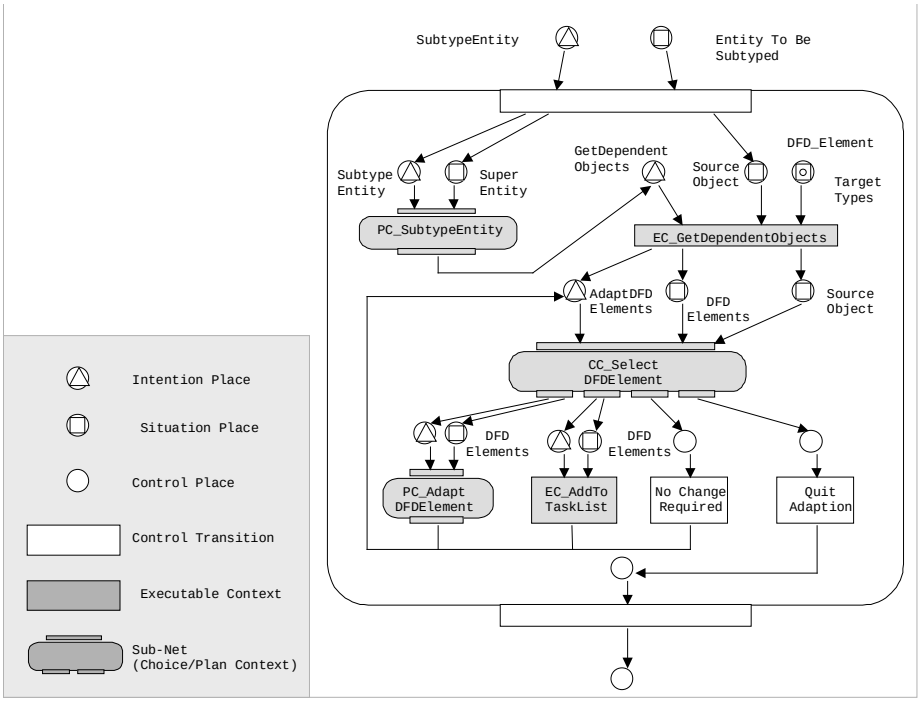

Figure 4 : SLANG definition of the plan context PC_SubtypeEntityAndAdjustDFD.

We will reuse this plan context to illustrate the assignment of executable and choice contexts to specific tool categories (Section 4.3) and to demonstrate the dynamic adaptation of the tool behavior (Section 8.4).

4.2 The Tool Meta Model: Defining Tool Capabilities

Representing methods and tools at a conceptual level is a prerequisite for comparing and mapping the services defined in the process/method model with the services offered by the tools of the environment. For achieving process-sensitive tools we propose to model tools not only in terms of the services provided (as in other PCE approaches), but also in terms of their graphical user interface and interaction capabilities.

In the following, we outline our tool meta model which was designed to facilitate an easy integration with the contextual process meta model. The cornerstone of the tool meta model is the concept *tool category*. By instantiating this concept the tool categories provided in the environment are defined such as an ER editor or DFD editor (Figure 5). The atomic services (actions) provided by a tool are defined as instantiation of the concept *action* like the action *CreateIsALink*. In addition, each atomic service (action) is related to the tool category via the *provides_action* association. For example, by instantiating this association one can define that the *ER_Editor* provides the action *CreateIsALink*. As a prerequisite for data integration the *input* and *output* parameters for each action have to be defined like the input of the *CreateIsALink* action (the *super* and *sub* entity type) or the output of the action (the created *IsALink;* cf. Figure 5).

The graphical presentation of the product parts is defined as instantiation of the association *displays* between a graphical shape and a product part provided by the tool. As depicted in Figure 5 it can be defined that an *Entity* (instance of *Product*) has to be displayed as a *Rectangle* (instance of *GraphicalShape*).

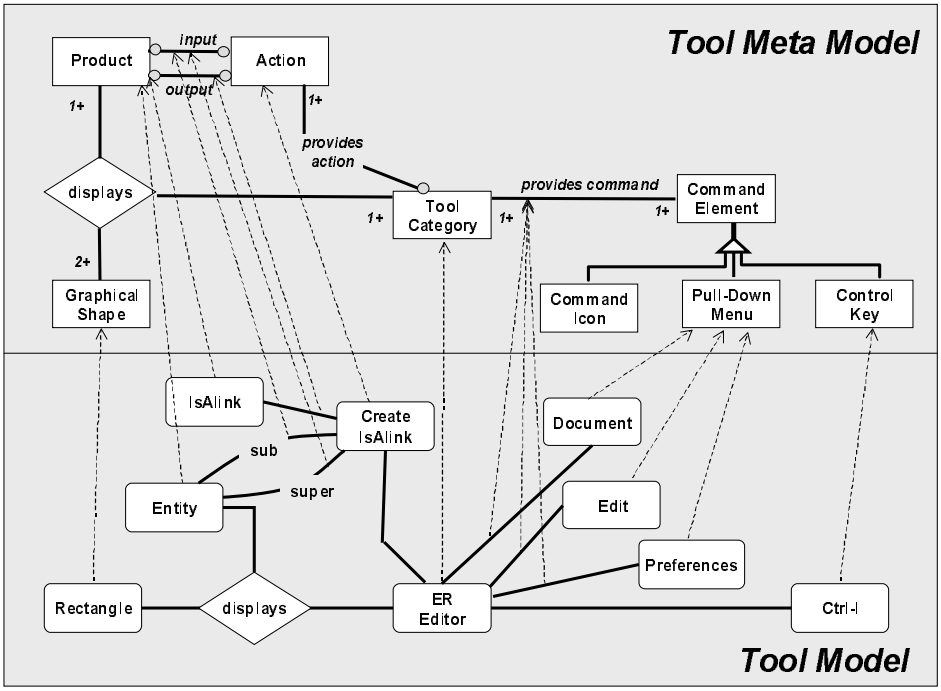

Figure 5 : The tool meta model.

Besides the capabilities of a tool for displaying product parts, also the interaction capabilities have to be defined. We assume that each tool enables the selection and de-selection of product parts and thus we do not model these interactions. In contrast, the *CommandElements* provided by a tool have to be explicitly defined. We distinguish three types of such capabilities, namely *PullDownMenu*, *ControlIcon*s and *ControlKey*s[3]. For example, the three pull-down menus provided by the ER editor (*Document*, *Edit*, *Preferences*) are modeled as instances of the concept *PullDownMenu* and related to the ER editor through instantiations of the *provides_command* association. Similar, the control key supported by the ER editor (*Ctrl-I*) is defined as instance of the class *ControlKey* and related to the ER editor.

[3] We currently support mainly window-based, interactive tools which enable user interactions via icons, menus, shortcuts and pointers. If other interaction facilities are required the tool model has to be adapted by defining corresponding concepts and relating them to the tool model and to the context definitions (see next section for details about the interrelations between the tool and the process definitions). However, as explained in detail in Section 6.2 the tools used in the environment must be able to interpret the models and adapt their behavior accordingly.

4.3 The Environment Meta Model: Interrelating Process and Tool Meta Models

The process meta model provides concepts for defining method fragments in terms of executable, choice and plan contexts. The tool meta model provides concepts for defining the capabilities of the tools available in the environment. By interrelating the tool and process meta models an integrated meta model, the so-called *environment meta model*, is formed which defines how and by whom a context has to be executed.

Since the tool meta model was designed with this interrelation in mind, the integration of the tool and process meta models is fairly easy. The integration can be achieved by defining three types of association between the two models (see Figure 6: dashed lines).

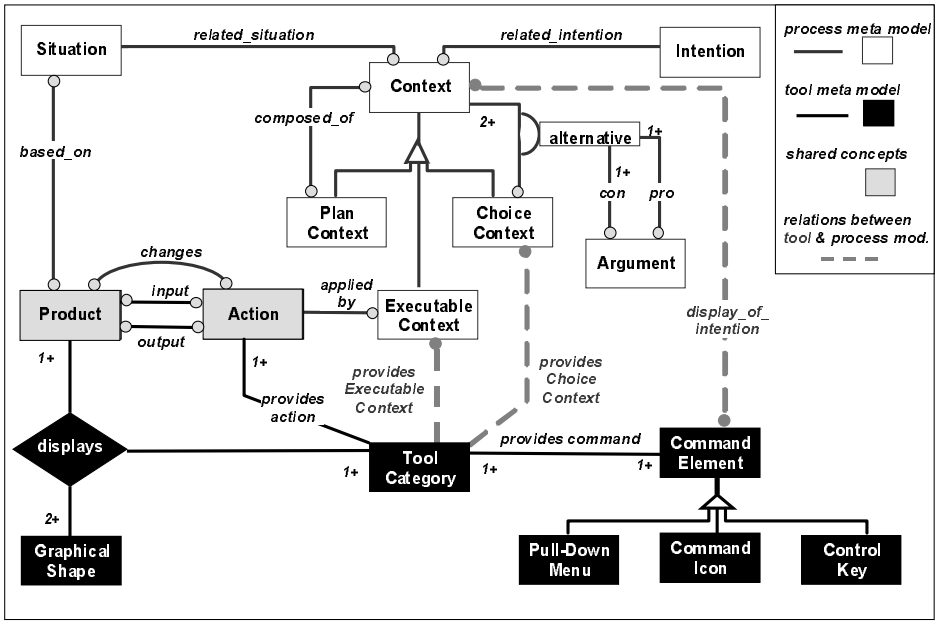

Figure 6 : Environment meta model: Integrating process and tool meta models.

4.3.1 Relating Tool Categories and Executable Contexts

Each executable context defined in the process model has to be associated with the tool category responsible for executing the context. This responsibility is represented as an instance of the association *provides_executable_context* (Figure 6). For example, if the executable context *EC_CreateEntity* is related to a tool category *ER_Editor* the ER editor has to perform the action associated in the process model with this context.

Given an executable context E, the associated action A and a set of tool categories T_1 - T_n, we distinguish three types of assignment:

- *automated assignment*: If there exists exactly one tool category T_i which offers the required action A, this tool category can automatically be associated to the executable context E;
- *choice of tool category*: If there exist two or more tool categories T_1 - T_n which offer the required action A, the method engineer must relate exactly one tool category with the executable context E;
- *lack of tool support*: If no tool category provides the required action, a new tool action has to be implemented in a tool and defined in the corresponding tool model, or the process model has to be changed.

For each assignment between a tool category and an executable context two consistency checks can be performed to ensure that the input and output defined for the tool action in the tool model corresponds with the process model definitions.

Constraint E1: Ensure that the output associations defined in the tool model between the action and the product parts are subsumed by the change associations defined for the action in the process model. Given an action A. Let P_o be the set of product parts related to A in the tool model using *output* associations and P_c the set of product parts related to A in the process model via *change* associations. Then, P_o must be a subset ofP_c.

Constraint E2: Ensure that all product parts defined as input for the action are subsumed by the situation of the executable context related to the action. Given an action A. Let P_i be the set of product parts related to A in the tool model via input associations. Let E be the executable context associated to the action A in the process model, S its situation and P_s the set of product parts defined for the situation. Then P_i must be a subset ofP_s.

Only if both checks are successful, i.e. if the input and output parameters defined in the tool model correspond with the process definitions, the tool category can be assigned to the executable context.

4.3.2 Relating Tool Categories and Choice Contexts

Each choice context has to be related to exactly one tool category by a *provides_choice_context* association. Thereby the tool category assigned to the choice context is made responsible for performing the choice context. Thereby a new guidance service is defined for the tool category.

In addition, for each context C_j defined as alternative of the choice context CC the presentation of the intention I related to the context C_j has to be defined. Since an intention like the intention *delete* can be associated with more than one context, for example, the contexts *deleteEntity* and *deleteAttribute*, a context dependent presentation of the intention is required. This is achieved by relating the context C_j to at least one command element using the *display_of_intention* association (see Figure 6).

As the relation of an executable context to a tool category, also the relation of a choice context to a tool category can be supported by consistency checks:

***Constraint C1**:* Ensure that the tool category can display all intentions associated with the alternatives of the choice contexts. Given a choice context CC which is related in the process model to a set of alternative contextsCA. For each context$C_x \in CA$, the tool category T associated (using the *provides_choice_context* association) with the choice context CC must be assigned to at least one command element (via a *provides_command* association) which is related (using a *display_of_intention* association) to the contextC_x.

***Constraint C2**:* Ensure that the tool category can display all product parts associated with the situations of all alternative contexts of the choice context: Given a choice context CC for which a set CA of alternative contexts is defined in the process model, and a set of product parts P_{CA} which subsumes all products related to any situation S which is related to a context$C_x \in CA$. If a tool category T is associated to the choice context CC then for all product parts $P_i \in P_{CA}$ a *displays* relation betweenP_i, a graphical shape G, and T must exist.

4.3.3 Environment Model: An Example

We illustrate the assignment of the contexts defined in the process model to the capabilities of the tools defined in the tool model using a small example (Figure 7). For readability, we depicted the name of the class of each instance in italics and brackets.

The upper part of Figure 7 depicts part of a process model where the choice context *CC_RefineEntity* is related to two alternative contexts, namely the executable context EC_CreateIsALink and the plan context *PC_SubtypeEntity*. In addition, the situations (*OneEntity*, *TwoEntities*) and the intentions (*CreateIsALink*, *SubtypeEntity*) of the two alternatives, the product parts (*Entity*) related to the situations and the action *CreateIsALink* of the executable context are defined in the process model.

The lower part of Figure 7 depicts part of the tool model where the tool category *ER_Editor* together with the supported control elements (the control key *Ctrl-I* and the pull down menu *Edit*) is defined. The tool category *ER_Editor* is further related to the concepts shared with the process model, namely to the action *CreateIsALink* using a *provides_action* association and to the product *Entity* via a *display_as* association.

In addition, the three associations defined in the environment meta model between concepts of the tool and process meta models have been instantiated (depicted as dashed lines in Figure 7):

- *Display_of_intention*: By instantiating this association the executable context *EC_Create_IsALink* is related to the control key *Ctrl-I* and the pull down menu *Edit*. Thereby the way of displaying the intention *CreateIsALink* related to the executable context is

defined. Similar, the plan context *PC_SubtypeEntity* is related to the pull down menu *Edit* (not shown in the figure);

- *Provides_choice_context*: The tool category *ER_Editor* is assigned to the choice context *CC_RefineEntity* by an instance of the *provides_choice_context* association. According to constraint C2 it must thus be ensured that the tool category *ER_Editor* can display the intentions of both alternatives of the choice context, namely the intention related to the contexts *EC_CreateIsALink* and *PC_SubtypeEntity*. The executable context *EC_Create_IsALink* is related to two command elements (the control key *Ctrl-I* and the pull down menu *edit*) which are both related to the tool category *ER_Editor* via a *provides_command* association. Thus constraint C2 is satisfied (for simplification the relation to the command elements of the plan context *PC_SubtypeEntity* are not shown in the figure). Constraint C1 is also satisfied, since all the product parts defined for the situations of both alternatives (in both cases the product type *Entity*) are related via a *displays* association to the tool category *ER_Editor* (see lower part of Figure 7);
- *Provides_executable_context*: The tool category *ER_Editor* can automatically be associated to the executable context *EC_CreateIsALink* by a *provides_executable_ context* association, since the executable context *EC_CreateIsALink* is related to the action *CreateIsALink* in the process model and this action is related to only one tool category, using the *provides_action* association, namely the tool category *ER_Editor*. In accordance to constraint E1, the *input* parameters defined in the tool model, (two product parts of the type *Entity*) are subsumed by the situation *TwoEntities* assigned to the executable context in the process model. Likewise the output (*IsALink*) is subsumed by the *change* association of the process model and thereby constraint E2 is satisfied.

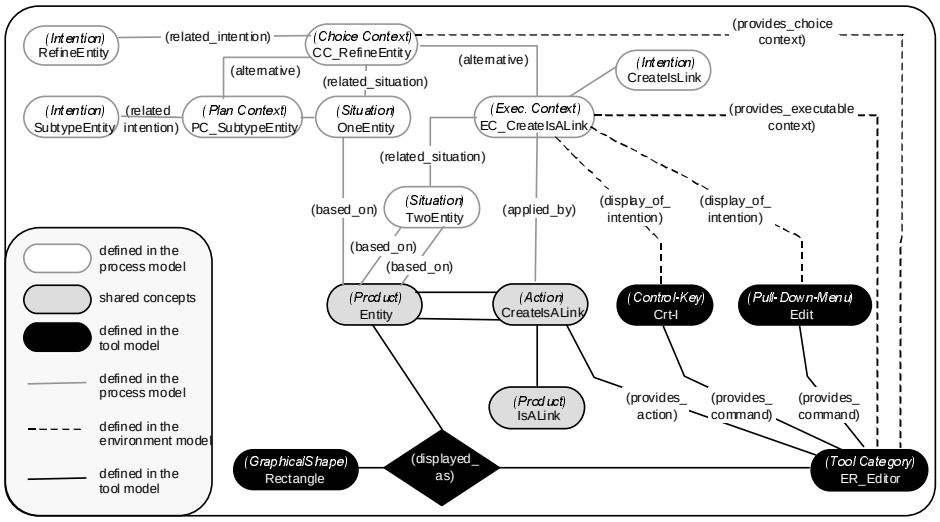

Figure 7: An environment model (simplified). <pohl7.ps, pohl7.tif >

4.4 Environment Meta Model: Contribution to Process-Integration Requirements

Representing both methods and tools at a conceptual level supports the method engineer in assigning the required tool functionality to the method definitions. Moreover, the above mentioned consistency constraints ensure correct assignments in the environment model. The environment model represents an important conceptual foundation for achieving a process-integration of the interactive tools in a PIE:

- *Data integration* is guaranteed by checking that the products subsumed by a situation correspond to the products used as input parameters of the tool actions.
- *Service integration* is achieved by the environment model through the interrelation of executable and choice contexts defined in the process model and the corresponding tool definitions.
- The *feedback information required* after context execution is inherently defined by the context types (in the case of executable contexts as output product types; in the case of choice context the contexts defined as alternatives).
- The foundation for the *invocation of method fragments* is established by the fact that a plan context can be related as an alternative to a choice context. This makes the tools aware of plan context definitions. The tools get to know the plan contexts which can be activated in a given process situation. Vice versa, the environment model empowers the enactment domain to invoke the tool responsible for executing a choice or executable context whenever such a context becomes active during the enactment of a plan context.
- The definition of the allowed graphical and interaction capabilities for each context lays the foundation for the *adaptation of the tool behavior*.

Within our PRIME framework, process-integration is mainly achieved through the appropriate interpretation of the environment model at run-time by all (!) three main components, namely the *tools*, the *enactment mechanism* and the *control integration mechanism*.

5 Synchronization of Performance and Enactment Domains

Synchronization of the performance and enactment domain is achieved by a process-aware control integration mechanism which directs message distribution based on the interpretation of the environment model (Section 5.1) and an interaction protocol which defines the behavior of the performance and enactment domain (Section 5.2).

5.1 Process-Aware Control Integration

The integrated environment model inherently assigns responsibilities for context execution. Executable and choice contexts are executed by the interactive engineering tools of the performance domain, while the process engine of the enactment domain is responsible for enacting the plan contexts. The interrelation of the three context types through choice contexts (via the *alternative* association) and plan contexts (via the *composed_of* association) requires interaction between the two domains.

In a process-integrated environment, message exchange should not be carried out in an uncontrolled manner. Instead, the control integration mechanism has to direct the message distribution according to the process definitions, respectively the definitions in the environment model in the case of PRIME. In other words, the control integration mechanism must be *process-aware*. For example, a service request from the enactment domain cannot be directed to an arbitrary tool providing the requested service. Instead, the request has to be directed to the tool responsible for performing the requested context according to the definitions in the environment model.

Technically, the interaction is carried out by message exchange which is typically employed by a control integration mechanism such as ToolTalk, COBRA, OLE/COM Automation. In PRIME, the information required for a process-aware control integration mechanism is represented in the environment model. Thus process-aware control integration can be achieved by implementing a trader on top of an existing control integration mechanism. The trader interprets the environment model and controls the message distribution accordingly. This ensures that during process enactment (and service brokering) service requests are directed to the tool assigned to the corresponding context in the environment model.

5.2 Interaction Protocol

The message types allowed depend on the current states of both the performance and the enactment domains. In contrast to conventional PCEs which are based on a simple client-server pattern, the context model induces more elaborate interaction patterns.

To ground message exchange on a solid basis we define an interaction protocol. The interaction protocol specifies the principal behavior of the enactment and performance domain in terms of the states and possible state transitions which are triggered by the delivery and receipt of messages. In addition, it defines the types of messages which can be exchanged between both domains.

We use the Statecharts formalism [71] for defining the interaction protocol. The behavior of the enactment domain is defined by the `Enactment-State` superstate, whereas the `Performance-`

`State` superstate specifies the behavior of the performance domain (Figure 8)[4] The coupling of both superstates is expressed by associating the transitions, which represent interactions between the two domains, with defined message types which are sent by one domain and received by the other.

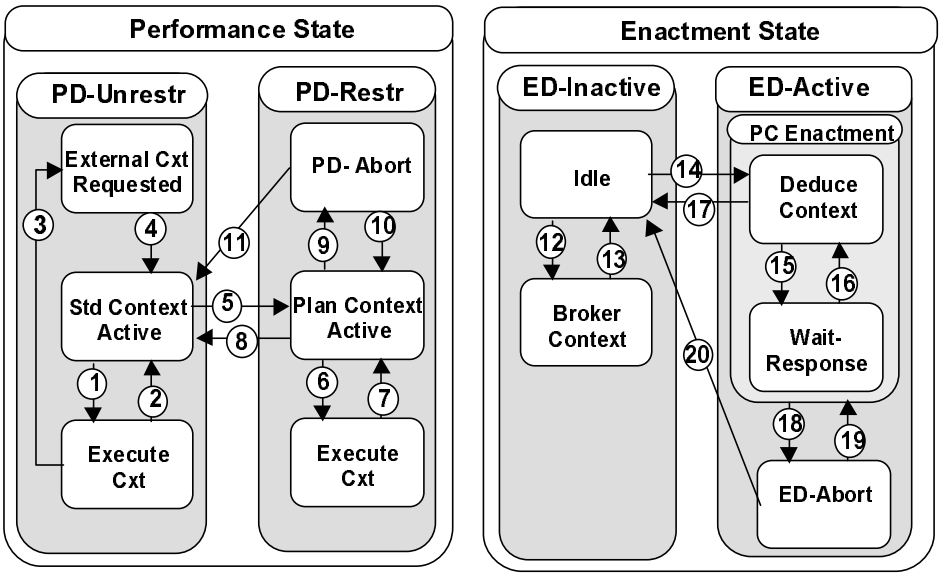

Figure 8: Statecharts defining the states of enactment and performance domains (simplified).

The contextual process model inspires the general distinction between the principal states of a process integrated environment:

1. *Restricted*: Process performance is governed by plan context enactment (Figure 8; `ED-Active` and `PD-Restr` superstates);
2. *Unrestricted*: Process performance is not restricted since no plan context is being enacted (Figure 8; `PD-Unrestr` and `ED-Inactive` superstates).

In the `ED-Inactive` superstate the process engine is essentially passive. In the `PD-Unrestr` superstate, more precisely in the `Std-Context-Active` state, a tool is able to perform any executable context and choice context provided by the tool[5].

If the user has activated an executable or choice context (see Section 8.4 for details) provided by the tool, transition 1 is performed and thus the `Execute-Cxt` state is entered. After executing the context the tool returns in the `Std-Context-Active` state by performing transition 2.

---

[4] For the sake of brevity, the Statecharts described in the following have been heavily simplified. A detailed description of our original interaction protocol, which defines about 30 different states and substates and 55 transitions in both domains as well as 15 message types, can be found in [4].

[5] Formally, the choices which are offered to the user in the PD-Unrestr state are modeled in the environment model as a choice context, called standard context.

If the user has activated a choice or executable context not provided by the tool, the tool sends a *Broker_Request* message (transition 3) to the control integration mechanism and enters the state `External_Cxt_Requested`. The control integration mechanism passes this message to the process engine. If the process engine is in the `ED_Active` state, the tool request is queued and the tool is informed. If the process engine is in the `ED_Inactive` state, it enters the `Broker-Context` state (transition 12) and requests the execution of the context by sending a message to the control integration mechanism (transition 13). The control integration mechanism determines the tool responsible for executing the requested context (according to the environment model) and passes the request to the tool.

If the user has activated a plan context, the tool sends an *Enactment Request* (transition 5) to the process engine and enters the `Plan-Context-Active` state. Receiving this message the process engine enters the `ED-Active` superstate (transition 14). After having locked all required tool resources (the locking sub-protocol is not shown here), the process engine determines in the `Deduce-Context` state the context to be performed next and sends a *Context_Request* message (transition 15) to the control integration mechanism; thereby it enters the `Wait-Response` state. The control integration mechanism determines the tool responsible for performing the context (according to the environment model) and sends the context execution request to the corresponding tool. By receiving the context request the tool changes in the `Execute-Cxt` state (transition 6) and returns, after having executed the context, in the `Wait-Request` state by sending a *Cxt_Feedback* message (transition 7). If an executable context was executed, the tool sends the feedback information to the process engine as defined by the output associations in the environment model. If a choice context was executed, the tool returns the selected alternative (context) to the process engine. After receiving the *Cxt_Feedback* message, the process engine enters the `Deduce-Context` state (transition 16) and determines the context to be executed next.

Process enactment stops if the enacted method fragment (plan context) is completed or if the user has requested to abort the context execution (see below). In both cases, the process engine releases all previously locked tools (not shown here in details). As a consequence, the process enactment domain enters the `ED-Inactive` state (transition 17 or 20), whereas the tools enter the `PD-Unrestr` superstate (transition 8 or 11).

The user can notify the process enactment domain about a process deviation whenever the enactment domain is active (`ED-Active` superstate). The user initiates such a request by choosing the abort-enactment context provided by each tool. In this case, the tool sends an *Abort_Request* message to the enactment domain (transition 9). Receiving this message (transition 18) the enactment domain enters the `ED-Abort` state and checks if the interruption of the user can be

applied, i.e. checks if the enactment can be aborted[6]. If the enactment can be aborted, the process engine sends an *Abort-OK* message and enters the `Idle` state (transition 20), otherwise it continues with the process enactment (transition 19). Correspondingly, the tool enters the `Std-Context Active` state (transition 11) or if it receives an *Abort Denied* message it returns back to the `Plan-Context-Active` state.

The interaction protocol sketched above defines the dynamic relationships between the performance domain and the enactment domain. Synchronization between the domains is achieved via special sub-protocols and message types, e.g. the messages exchanged during the locking phase before actual enactment starts. This ensures that the relevant tool resources are available and ready to accept the requests coming from the enactment domain.

The interaction protocol (together with the environment model) supports both *reactive* and *proactive* process enactment styles which are extensively demanded in literature, e.g. [50]. Reactive control means that the process performer can operate freely on his tools and at some point initiate a request to the enactment domain. Proactive control means that the enactment domain initiates the operations and governs the possible user choices in the performance domain. The *Unrestricted* and *Restricted* states of both domains reflect these two modes.

6 The PRIME Implementation Framework

The environment model (Section 4.3), the process-aware control integration mechanism and the interaction protocol (Section 5) provide the conceptual foundations for establishing a process-integration in engineering environments.

To facilitate the development of a process-integrated environment (PIE) we developed the PRIME framework which meets the technical requirements for PIEs derived from those conceptual foundations (Section 6.1). The main architectural components of PRIME are the generic tool architecture (Section 6.2) and the generic enactment architecture (Section 6.3). The PRIME components have been implemented as a reusable, object-oriented implementation framework (Section 6.4).

---

[6] Since arbitrary interruption of context execution can cause data inconsistencies, we require that the method engineer defines in the process model the situation in which process deviations are allowed, i.e. in which the enactment can be aborted, and/or that she or he defines additional actions to be applied for enabling a process deviation. The user can only abort the enactment of a context if the process model allows the abortion. "Backtrack" mechanisms which, in the case of an Abort Request, set the enactment back to a situation in which no data inconsistencies are caused by the interruption of the process enactment, are an open research issue.

6.1 Requirements for the PRIME Components

The components of a process-integrated environment have to consider the definitions in the environment model for a model-conform process performance and must obey the interaction protocol for synchronizing the states of the enactment and the performance domains. This poses several requirements on the three main components of PRIME.

The *control integration mechanism* must be process-aware. It must distribute context requests and feedback messages according to the context assignments expressed in the environment model.

The *tools of the performance domain* have to fulfill the following requirements:

RT1 *Execution of executable contexts according to environment model*: The activation of an executable context in a tool must result in the invocation of the tool action related to the executable context in the environment model. In addition, the results of executing this action must be passed to the context invoker;

RT2 *Execution of choice contexts according to environment model*: The activation of a choice context must result in a user interface adaptation of the tools according to the definition of the choice context in the environment model. The tool must adjust the products and command elements displayed at the user interface according to the context definition and highlight the selectable products and command elements;

RT3 *Detection of context activation according to environment model*: The tool must be able to compare the product parts and command elements selected by the user with the context definition.

RT4 *Synchronization with enactment domain according to interaction protocol*: The tool must exchange messages with the enactment domain in accordance with the interaction protocol defined in Section 5.2.

The *enactment mechanism* has to fulfill the following requirements:

RE1 *Enactment of plan context definitions*: The enactment mechanism has to interpret an activated plan context. It has to deduce the context to be executed next and it has to initiate its execution. In addition, the enactment mechanism has to interpret the feedback information obtained from the context execution for determining the context to be executed next;

RE2 *Synchronization with performance domain according to interaction protocol*: The enactment mechanism must exchange messages with the performance domain in accordance with the interaction protocol defined in Section 5.2.

6.2 Generic Tool Architecture

We have designed a generic tool architecture which fulfills the requirements RT1 - RT4 outlined above. Figure 9 depicts the main architectural components and their relations of the generic tool architecture of PRIME. The generic tool architecture has two central subsystems: The StateManager (Section 6.2.1) and the ContextManager (Section 6.2.2).

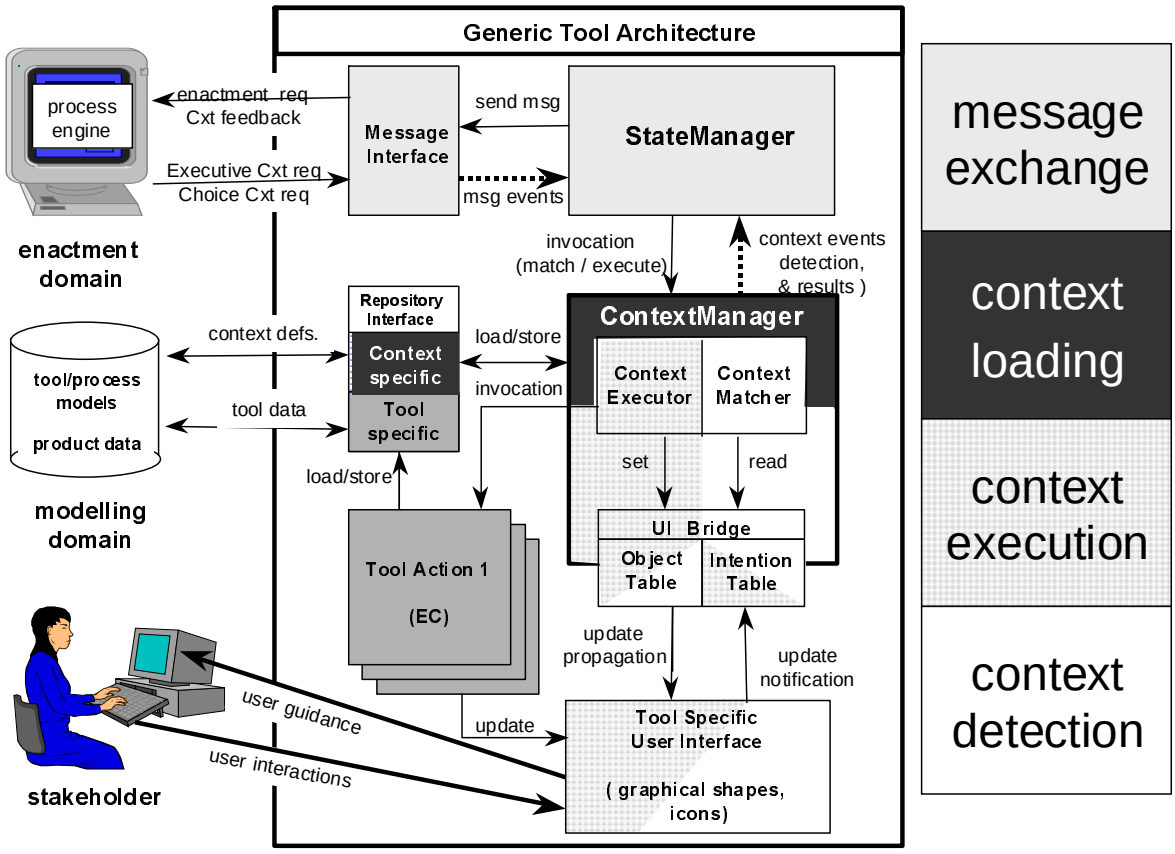

Figure 9: Generic tool architecture. <pohl9.ps, pohl9.tif >

6.2.1 The StateManager

The *StateManager* subsystem (upper right part of Figure 9) ensures that the message exchange with the process engine is carried out in accordance with the interaction protocol described in Section 5.2 (requirement RT4). It governs the overall control flow in the tool and maintains the tool state in response to events received. There are two types of events:

- *External events* are initiated through the receipt of a message from the enactment domain such as a request from the enactment domain to execute a context during plan context enactment. The message receipt and delivery is handled by the MessageInterface which is closely connected to the StateManager.

- *Internal events* are either generated by the ContextExecutor for reporting the results of a context execution or by the ContextMatcher after the identification of a context to be executed. If the identified context is a plan context or a context provided by another tool (according to the environment model), the StateManager sends an enactment request together with the situation data via the MessageInterface to the enactment domain. Otherwise the StateManager requests the execution of the context from the ContextExecutor.

6.2.2 The ContextManager

The task of the *ContextManager* is threefold. During the start-up phase the ContextManager retrieves all context definitions specified in the environment model for the tool category and stores them in a context cache. The *ContextExecutor* subsystem is responsible for adjusting the tool behavior and for providing user guidance according to the environment model (requirement RT1 and RT2). The *ContextMatcher* subsystem is responsible for the identification of method fragments (requirement RT3).

The ContextExecutor: Adaptation of Tool Behavior

The ContextExecutor controls the execution of choice and executable contexts.

If the StateManager requests the execution of an automated service (executable context), the ContextExecutor invokes the tool action associated to the executable context in the environment model. The situation data obtained with the context execution request are mapped to the input parameters. This is facilitated by defining the input parameters of the actions in the tool models using the same product types as the ones used for defining the situation types associated to the contexts in the process model.

If the StateManager requests the execution of a guidance service (choice context), the ContextExecutor adapts the user interface of the tool according to the definition of the choice context and the current situation data. More precisely, in the command region of the user interface only those menu items and icons are displayed which are associated to an alternative of the choice context and thus only the intentions associated to an alternative context are displayed. All other menu items and icons not related to an intention of an alternative of the choice context become unselectable. In the product region, all products corresponding to the situation data of the choice context are *highlighted* to draw user attention on them. Furthermore, all products which may contribute to a situation of an alternative context are displayed as *selectable*, whereas all other product parts become *unselectable*. To support the choice of an alternative, the user can always initiate the display of the arguments associated with the alternatives of a choice context. On user request, the ContextExecutor displays the pros and cons for each alternative of the choice context defined in the environment model in a special guidance window (see Section 8.4 for an example).

In the following we illustrate the adaptation of the tool behavior during the execution of a choice context. The right part of Figure 10 shows an entity relationship (ER) editor which currently executes the choice context *CC_RefineEntity* with the entity type *publication* as actual situation data. The left part depicts parts of the corresponding environment model.

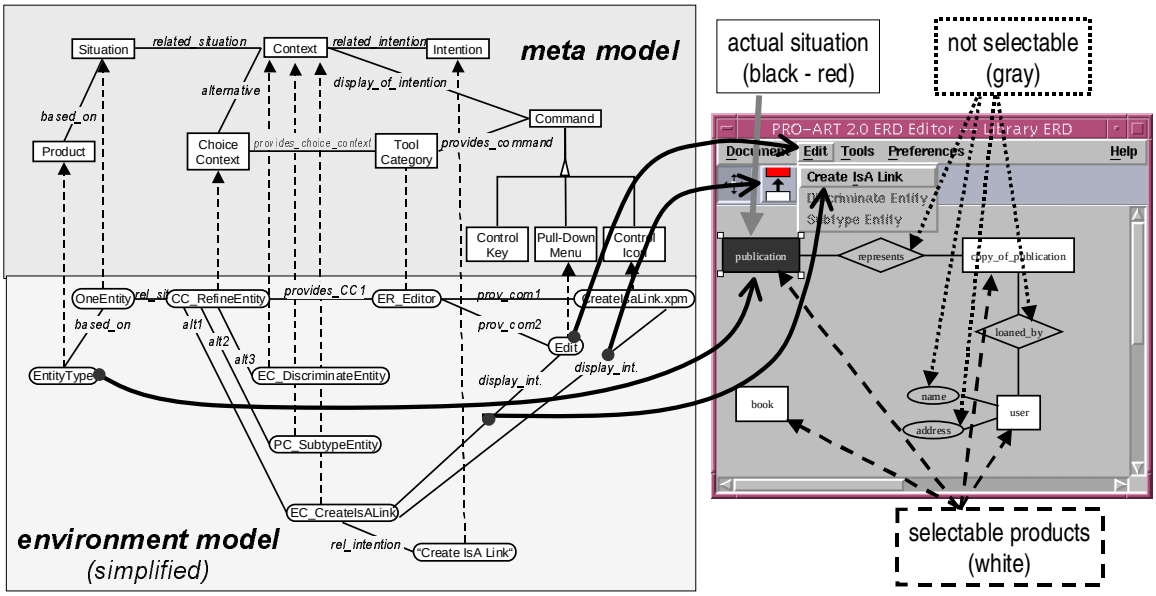

Figure 10: Adaptation of tool behavior according to the environment model.

The choice context *CC_RefineEntity* defines three different alternatives (*EC_CreateIs-ALink*, *EC_DiscriminateAttribute*, and *PC_SubtypeEntity*) for the refinement of an entity. The menu items displayed correspond to the definitions in the environment model. For example, according to the associations specified in the environment model, the intention of the alternative context *EC_CreateIsALink* appears as menu item in the *Edit* pull-down menu and as icon in the icon bar using the bitmap *CreateIsALink.xpm* in the ER editor (Figure 10).

In the product area, the entity type *proceedings* representing the actual situation data is highlighted. According to the environment model, the situations of all three alternative contexts are only based on entity types. Consequently, the ContextExecutor has marked all other objects as unselectable (displayed in gray) and thus only the entity types *book*, *copy_of_publication*, and *user* are selectable (displayed in white).

The ContextMatcher: Invocation of Method Fragments

During the execution of a choice context, the user selects and deselects product parts and activates command elements. The task of the ContextMatcher is to compare the user interactions with the context definitions which are defined as alternatives of choice context active. More precisely, it matches the activated command elements with the intentions associated to an alternative context

of the choice context and the selected product parts with the situations of the alternative contexts[7]. Whenever the selected product parts and the intentions match with the definition of an alternative context, the ContextMatcher requests the execution of the context from the StateManager.

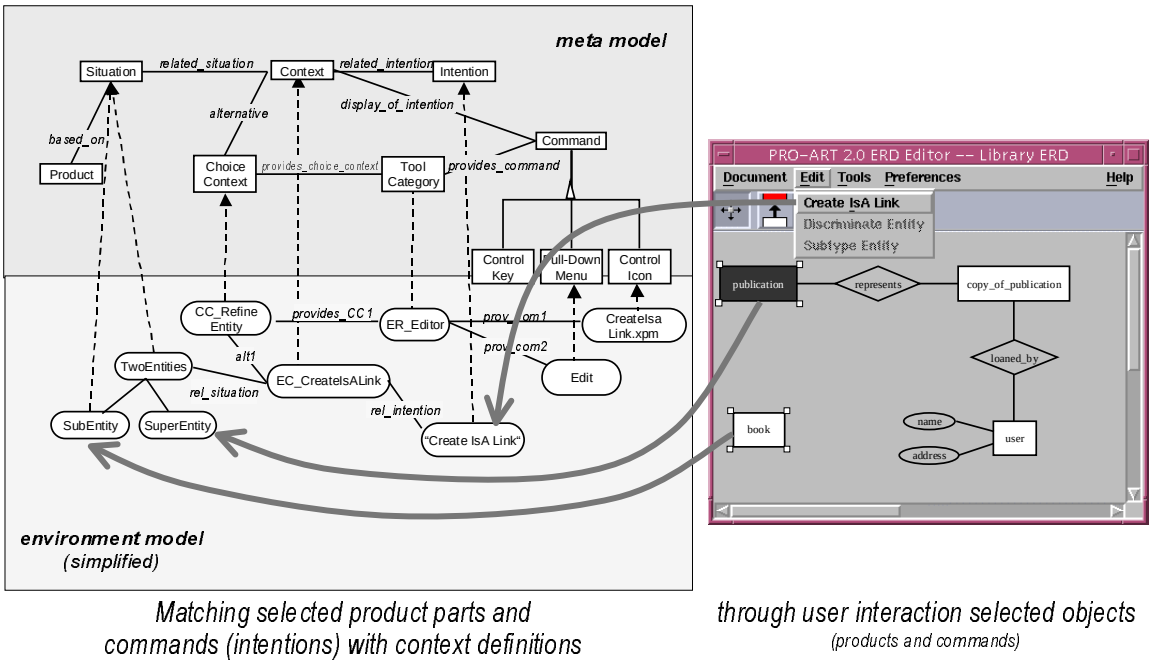

Figure 11: Matching a context.

Figure 11 illustrates the context matching. The ER editor is in the choice context *CC_RefineEntity*. After the user has selected the menu item *Create-IsA-Link*, the ContextMatcher compares the selected product parts (the two entities *publication* and *book*) and the intention associated with the selected menu item with the alternative contexts of the choice context defined in the environment model[8]. Comparing the selected product parts and command elements, the ContextManager detects that the selected items match with the definition of the executable context *EC_CreateIsALink*.

According to the environment model this context is provided by the ER editor itself. Thus, the StateManager passes the context execution request to the ContextExecutor of the ER editor which executes the context according to its definition.

Note that the execution of any context is based on the interpretation of the environment model by the ContextExecutor and the ContextMatcher. Changes in the method definitions require thus no

---

[7] The matcher currently applies a best fit approach. It thus associates a situation slot with the most specific selected product part.

[8] For efficiency reasons the matching is performed whenever a command element (intention) has been selected by the user.

re-programming and can mostly be achieved on a modeling level (see Section 9 for a detailed discussion about the integration of method changes).

6.3 Generic Enactment Architecture

The generic enactment architecture drives process enactment by interpreting the process relevant parts of the environment model. The architecture handles enactment requests of the performance domain and initiates the enactment of the requested method fragments. Similarly to the tool architecture, the enactment architecture consists of two central components: the *ED_StateManager* and the *ED_ContextManager*.

The *ED_StateManager* realizes the interaction protocol described in Section 5.2 from the perspective of the enactment domain. It controls the current enactment state according to the Statechart defined for the enactment domain (Section 5.2; requirement RE2).

During the enactment of a plan context, the *ED_ContextManager* is responsible for deducing the context to be performed by interpreting the plan context definition. It also initiates the execution of the deduced context (requirement RE1).

The generic enactment architecture was designed with the purpose of enabling experimentation with different process enactment languages. For an easy integration of existing enactment mechanisms (e.g. for "plugging" in a SLANG net interpreter) the *ED_ContextManager* offers generic interfaces which provide functions to

- Inform the enactment mechanism about the activation of a plan context;
- Send a context execution request (executable or choice context) to the performance domain;
- Process the context execution results received from the performance domain.

From the dynamic point of view, the invocation of these functions is encapsulated in a single state, namely the `Deduce-Context` state (see Section 5.2).

6.4 Implementation of the PRIME Framework

The generic parts of the architecture described in Section 6.2 and Section 6.3 have been implemented as an object-oriented implementation framework in C++ on two different platforms (Sun Solaris Unix and Windows NT). Figure 12 provides an overview of the *PRIME implementation framework*. The white parts depict the generic components of the framework which are re-used without any adaptations for implementing a process-integrated modeling environment for a particular application domain such as requirements engineering or chemical engineering. The black parts denote application domain specific components.

The *process repository* stores the environment model (the product, process and tool models and the integration associations). It has been implemented on top of a relational DBMS (Sybase 11

server). The environment meta model presented in Section 4 has been transformed into a relational schema consisting of 27 tables.

The four *meta-modeling tools* facilitate the creation and maintenance of product models, method fragments, tool models, and their interrelations in the repository.

The *generic tool framework* of the performance domain facilitates the implementation of interactive, process-integrated tools. It provides libraries for context management, state management, and user interface adaptation. We have carefully separated the user interface library from the context and state management libraries to allow an easy adaptation of the framework to another user interface toolkit. The context management library consists of a repository layer for loading context definitions from the repository and components for matching and executing contexts. The state management library provides components for handling the receipt and delivery of messages (including parsing and unparsing of messages) and for maintaining the current tool state according to the interaction protocol. Both libraries comprise about 70,000 lines of C++ code. The user interface library provides a user interface bridge for the ILOG Views toolkit. The components of this library map, among others, command elements defined in the environment model to ILOG Views specific menu classes. The user interface library comprises about 15,000 lines of C++ code.

Conformance to common usability standards is largely ensured by the ILOG Views toolkit which supports both Motif and Windows look-and-feel. Moreover, the definition of common menus, short-keys, command icons, and standard shapes ensures that similar functionality can be uniformly accessed in all tools.

We initially expected a general degradation of tool performance due to the processing of each object (de-)selection and menu activation by the context matcher. But even in tests with more than 200 product objects and choice contexts with more than 100 alternative contexts no noticeable increase in the response time to user events could be observed. This is mostly due to the fact that after loading the tool relevant context definitions from the process repository during the tool startup phase, they are maintained in a context cache within the tool. The loading of the context definitions, however, slightly increases the startup time (about one second in average; depending on the number of contexts defined for a tool).

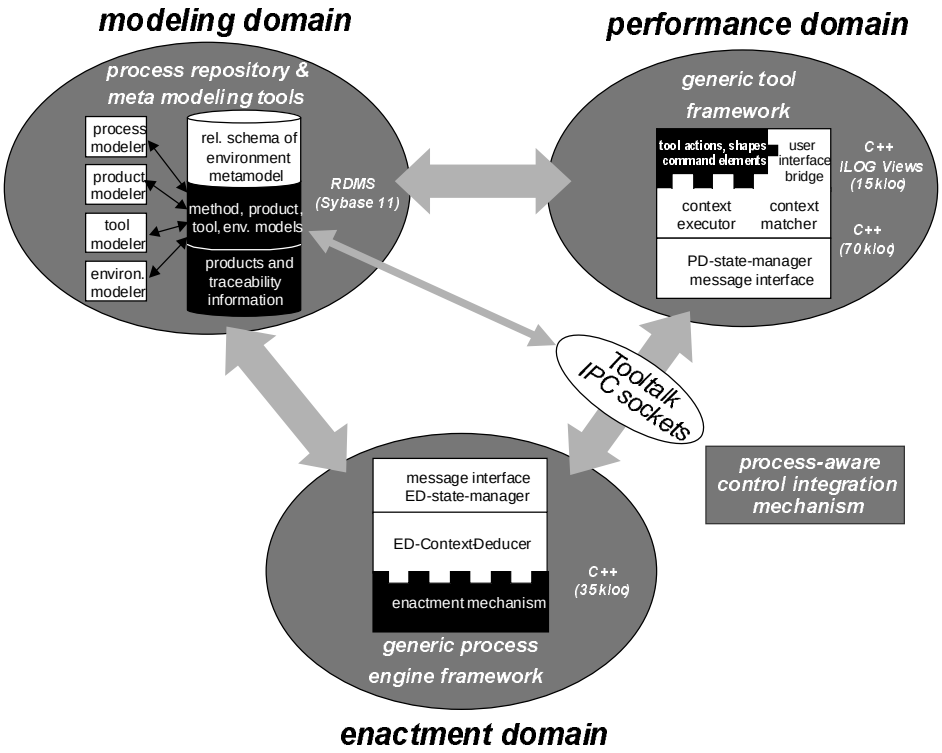

Figure 12: Implementation of the PRIME framework.

The *generic process engine* framework of the enactment domain facilitates integration of an enactment mechanism for a given plan context definition language by providing abstract base classes for context deduction. To embed a specific enactment mechanism these base classes have to be specialized. The implementation of the generic process engine framework shares considerable parts with the generic tool framework such as the base classes of the state manager and the message handling components. Altogether, the implementation framework for the enactment mechanism comprises about 35,000 lines of C++ code.

The *process-aware control integration mechanism* has been realized using the standard socket library (both for Solaris Unix and Windows NT), and in an alternative implementation using the more convenient services provided by SUN's ToolTalk [54] (only Solaris Unix). The process-aware trader defined on top of these mechanisms maintains knowledge about the running tool instances. It controls the message exchange between the performance and enactment domain based on this knowledge and the interpretation of the environment model. For example, it sends a context execution request to the tool responsible for executing the requested context.

7 Integration of Legacy Tools

To achieve a process-integration of a legacy tool, the requirements RT1 - RT4 described in Section 6.1 must be fulfilled. Since most existing tools do not meet these requirements,

appropriate wrappers must be designed and implemented. The wrappers make use of the application programming interfaces (APIs) provided by the tools and add additional functionality to a tool which achieves a process-integration of the legacy tool by fulfilling the requirements RT1 - RT4.

We first elaborate on the kinds of APIs a legacy tool must provide in order to be fully process integrable (Section 7.1). We then describe an extension of the generic tool architecture of the PRIME implementation framework which facilitates the integration of legacy tools (Section 7.2). We illustrate the use of the extended architecture to realize the process-integration of VISIO, a commercial CAD tool, in the PRIME-based environment TECHMOD (Section 7.3).

7.1 Required Application Programming Interfaces (APIs)

The tool requirements sketched in Section 6.1 require that a legacy tool has to provide certain APIs (application programming interfaces) to be process integrable:

A1 A *service invocation API* required for activating the actions provided by the tool including passing of the actual parameters on which the action should be performed (requirement RT1);

A2 A *feedback information API* required for accessing the results obtained from executing an action (requirement RT1);

A3 A *command element API* required for introducing new/additional command elements like menu options or graphical icons defined in the environment model (requirement RT2);

A4 A *product display API* required for highlighting the product parts constituting the actual situation of a choice context (requirement RT2);

A5 A *selectability API* required for adapting the user interface of the tool according to the definitions of the active choice context and its alternative contexts (requirement RT2);

A6 A *selection notification API* required for obtaining notifications about user selections of products and command elements. This is a prerequisite for matching the user interactions with the context definitions and thereby supporting the activation of a predefined context (requirement RT3);

Achieving a synchronization with the enactment domain (requirement RT4) does not require a special tool API. Definition conform synchronization can be ensured by a wrapper which uses the APIs A1 - A6 in accordance to the interaction protocol definition.

If a legacy tool provides the six APIs sketched above, a process-integration of the legacy tool can be achieved by designing and implementing appropriate wrappers.

The process-integration of legacy tools which only provide a subset of the required APIs is by far not easy. To support the process-integration of such tools, a more comprehensive framework is required. Such a framework should support the process-integration along four major lines:

- Providing a check list and criteria for assessing the degree of process-integration which can be achieved for a legacy tool based on the APIs provided by the tool and its technical implementations;
- Providing generic wrapper components which can be reused (adapted) for achieving a process-integration of a legacy tool;
- Defining the relations between the generic wrappers and the assess criteria to support the selection of the wrappers based on the assessment of the tool;
- Relating those wrappers to the generic tool architecture to enable as much reuse as possible.

Establishing a comprehensive framework for the process-integration of any kind of legacy tool is a major future research activity. The definition of such a framework could start from the solution provided for legacy tools which provide the required APIs. In the following we describe the process-integration of legacy tools which provide the six APIs defined above and can thus can be fully process-integrated.

7.2 Integrating Legacy Tools Using the Generic Tool Architecture

There are two main alternatives for designing and implementing the wrappers. The wrappers can be designed in a way which foresees a direct interaction between the wrappers and the enactment domain. Alternatively, the wrappers can be designed to wrap the legacy tools into the generic tool architecture provided by the PRIME implementation framework. In this case, the enactment domain interacts with the generic tool architecture which itself wraps the legacy tool. The latter alternative stands to reason since the functionality to be provided by the wrappers is, to a large degree, covered by the functionality provided by the generic tool architecture. Examples are the consideration of the interaction protocol, the context matching and the context execution.

The main problem hindering a simple wrapping of legacy tools into the generic tool architecture is that even if a tool provides the APIs with the functionality sketched above, the signature of the functions provided by the various APIs and the invocation protocols for using the APIs significantly differ between legacy tools. We have thus investigated in an extension of the generic tool architecture which facilitates the process-integration of legacy tools and minimizes the implementation efforts and the legacy tool specific influence on the generic tool architecture.

As depicted in Figure 13 we extended the generic tool architecture of the PRIME implementation framework with two adapter layers which encapsulate the generic parts of the architecture (StateManager, ContextExecutor, ContextMatcher, Object Table, Intention Table). Technically, the adapter layers are realized as a set of classes whose abstract interfaces (virtual methods) are used by the generic parts of the PRIME implementation framework.

To achieve a process-integration of a legacy tool, these classes are specialized by overwriting the virtual methods by specific methods which bridge the functionality provided by the adapter layers and the APIs provided by the legacy tool.

The *ActionAdapter* (lower left part of Figure 13) maps the actions defined in the tool model to the service invocations provided by the legacy tool. Consequently it makes use of the *service invocation* and *feedback information* APIs (A1 and A2). More precisely, the adapter classes between the tool actions and the context executor are specialized using the corresponding APIs to assign each executable context defined for the legacy tool to its service. A main task of the adapter class is thereby to map the product data embedded in a situation instance to the data format of the input parameters required by the service invocation API of the legacy tool and vice versa.

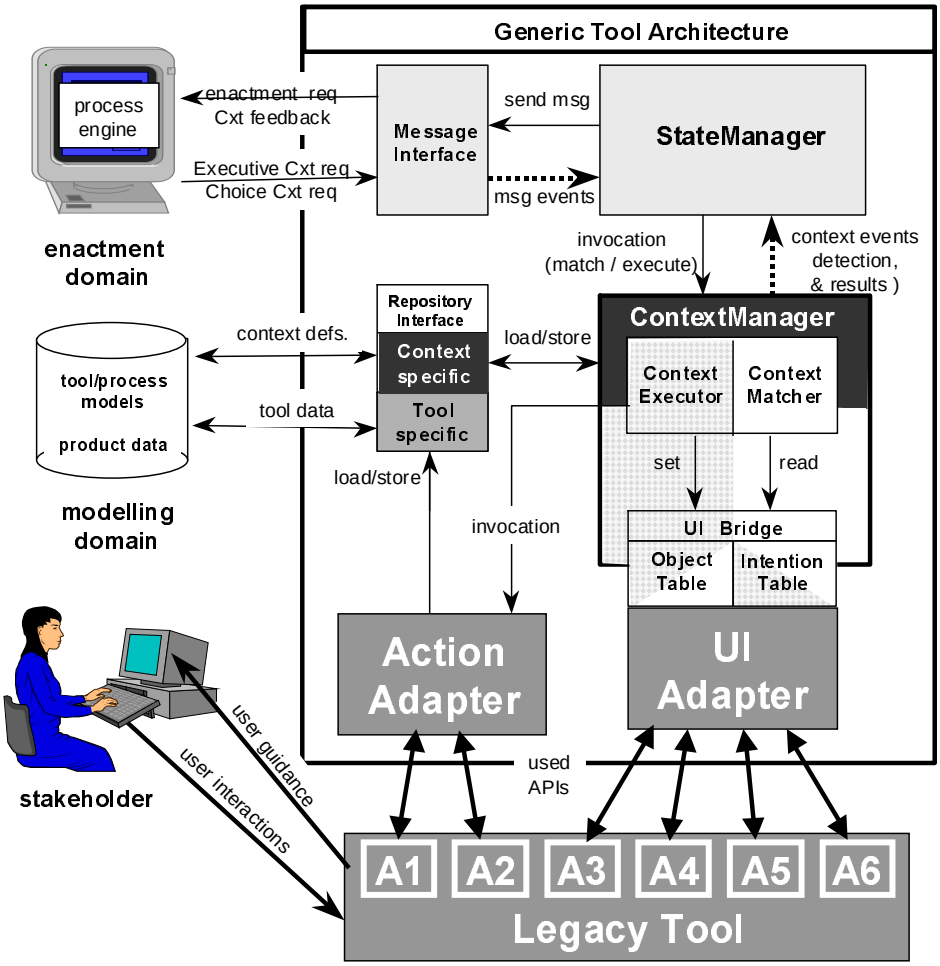

Figure 13: Adding adapter layers to the generic tool architecture for integrating legacy tools.

The *UserInterfaceAdapter* (lower right part of Figure 13) bridges the object and intention table of the generic tool architecture with the user interface(s) of the legacy tool. For this purpose, the APIs A3 - A6 of the legacy tools are used in the specialized adapter classes.

The UserInterfaceAdapter ensures that the user interface of the tools corresponds with the actual status of the intention and object tables. More precisely, the adapter ensures that in the user interface only the command elements of the intention table can be activated and that all products which may contribute to a situation of an alternative context are displayed as selectable, whereas all other products become unselectable. Moreover, it ensures that all products corresponding to the situation data are highlighted. Since the object and intention table is updated by the Context Executor according to the choice context definition and the actual situation data, the UserInterfaceAdapter indirectly guarantees that the user interface of the tool is adapted according to the context definition and the actual situation data. Moreover, the UserInterfaceAdapter ensures that the user interactions (selection and de-selection of products and intentions) lead to an update of the intention and object tables. Thereby the detection and activation of a predefined context is achieved, since the Context Matcher compares the intention and object table with the context definition.

To summarize, the extended generic tool architecture of the PRIME implementation framework facilitates to a large degree the process-integration of legacy tools which provide the required APIs:

- The implementation effort for building a wrapper is significantly reduced. Only the classes of the adapter layers have to be specialized for using the specific APIs of the legacy tool at hand;
- The tool builder responsible for wrapping a legacy tool does not have to care about possible interdependencies between the use of the individual tool APIs and their interplay with the enactment architecture since this is already defined and realized by the generic tool architecture;
- The tool builder responsible for wrapping a legacy tool does not have to care about the correct consideration of the environment model during context detection and context execution. Similarly he does not have to worry about implementing wrappers which ensure a correct tool behavior as defined in the synchronization protocol (correct message and event handling). All this is guaranteed through the wrapping of the tool into the extended generic tool architecture of the PRIME implementation framework.

7.3 Integrating Legacy Tools Using the Generic Tool Architecture: An Example

The goal of the Collaborative Research Center SFB-476 IMPROVE (funded by the Deutsche Forschungsgemeinschaft), is to establish computer-based support for the design of chemical processes and chemical plants for producing new chemical products in the large scale [72], [73]. In this project, integrated method guidance for the engineers defining and simulating conceptual models of chemical processes and plants is achieved through a PRIME-based environment, called TECHMOD (Traced Engineering of CHemical process MODels) [74], [76]. Within the SFB-476 IMPROVE many commercial tools are used to support the design and construction of a chemical

plant including CAD tools, simulators and model builders. To achieve integrated method guidance those tools must be "process-integrated" with the TECHMOD environment.

We have experimented with the integration of three legacy tools. One of them, VISIO, offers all six APIs defined above. In the following we sketch the process-integration of VISIO into the TECHMOD environment and illustrate the use of the extended generic tool architecture.

VISIO is a CAD tool which provides special graphical icons and functions for constructing flow sheet diagrams. Flow sheets are a common abstraction used in the chemical industry to describe chemical processes. VISIO runs under WINDOWS 95/NT and provides OLE/COM application programming interfaces for extensions.

To achieve a process-integration of VISIO we first defined the services and the command elements provided by VISIO using the concepts of the tool meta model. The resulting VISIO tool model was integrated with the method definitions. There was no difference in modeling the capabilities of VISIO in comparison to the definition of the capabilities of a new tool, implemented using the generic tool architecture.

For embedding VISIO in the generic tool architecture we specialized the ActionAdapter and the UserInterfaceAdapter. The specialization (implementation) of the adapters was straightforward. The tool action invocations initiated by the context executor were mapped by a specific adapter class to the services provided by VISIO. Similarly, the user interface facilities of VISIO were assigned to the user interface bridge. However, two unexpected “conceptual” problems (1 and 2) and three technical problems (3 to 5) worth mentioning occurred:

1. Each VISIO action could potentially occur in 14 different menu bars, each of which had consequently to be controlled by the wrapper. The consideration of all possible menu bars obviously caused additional effort to "wrap" the intention table of the generic tool architecture with the command and selectability APIs of VISIO.

2. Some VISIO actions could be activated by drag-and-drop mechanisms. Since the PRIME implementation framework does not support drag-and-drop mechanisms, we have deactivated the drag-and-drop mechanism of VISIO[9].

3. We originally planned to run VISIO and the PRIME implementation framework as separate operating system processes where the former was invoked through the COM/WIN32 API as Automation Server by the latter which acted as Automation Client. However, in this mode VISIO (for some non-obvious reasons) does not provide the possibility to register for notifications of menu selection events. The VISIO API (which is required for context matching) is only available for "in-process" extensions of VISIO. The API can thus only be used if the PRIME implementation framework is linked to VISIO during start-up as dynamic

[9] We are currently extending the tool meta model and the implementation framework to support command activation using drag-and-drop mechanism.

link library (DLL). Consequently, we had to transform the formerly static libraries of the PRIME implementation framework into DLLs;

4. As a consequence of melting VISIO and the PRIME implementation framework there were now two event loops within one operating system process: One of VISIO and the other one of the PRIME implementation framework (for handling message events from the enactment domain). Since these event loops initially interfered with each other, the PRIME event loop had to be adjusted;
5. VISIO assumes that layout information of diagrams is stored in normal files in a proprietary data format whereas PRIME stores the logical product information of the flow sheet diagram being built in the process repository. Consequently, the Action Adapter has to establish and to maintain links between the layout data in the VISIO files and the corresponding product data in our process repository.

The technical problems closely correspond to the problems encountered when integrating several implementation frameworks as described in [76].

To summarize, the process-integration of VISIO was facilitated to a large degree by wrapping it with the generic tool architecture of the PRIME implementation framework. Since VISIO provided the required APIs the wrapping of the tool services and the user interface was straightforward, despite of the technical problems.

8 Building PRIME-Based Environments

We have identified six steps for building a PRIME-based process-integrated environment (PIE). In the following we elaborate on each step (see Table I for an overview).

Table I. Six steps for building a PRIME-based environment.

Step	Description	Modeling Level	Implementation Level
1	Choose one or more plan context definition languages	X	
2	Define method guidance in the process model	X	
3	Define tool capabilities of the legacy and new tools in the tool model	X	
4	Define the environment model	X	
5	Implement domain-specific tool functionality and/or wrapper for legacy tools		X
6	Integrate required enactment mechanism(s)		X

8.1 Six Step Procedure for Building a PRIME-based Environment

Step 1 - Choose One or More Plan Context Definition Languages: A plan context consists of a set of contexts of any type. It defines a sequence of the embedded contexts by a control flow. For defining the control flows one or more suitable languages have to be chosen. The choice of a language depends on the expressiveness required. For example, a finite-state machine language might be chosen if mainly sequential invocation should be defined, or an imperative programming language might be chosen if many branches and conditional loops have to be defined.

To enable the embedding of a context in a plan context, templates have to be defined in the chosen language(s) for representing the three context types, respectively their interfaces (situation and intention). For example, in the case of the Petri-Net based language SLANG special place types for expressing the situation and intention and sub-net templates for each of the three context types have been introduced (see Section 4.1.2 for details).

Step 2 - Define Method Guidance in the Process Model: The method engineer defines the method guidance using the concepts provided by the contextual process model. After defining the product model, the method engineer identifies the relevant product constellations (modeled as situations). In addition, she or he defines the goals to be achieved (modeled as intentions). By assigning an intention to a situation she or he defines the contexts to be supported. Next the method engineer specifies the guidance to be offered when the context is activated by specifying the implementation for each context.

If the context is fine-grained enough to be implemented as a single action, the context is defined as executable context and related to the action to be executed.

Decision points where user intervention is required are expressed by choice contexts. In addition, the method engineer defines the pro and con arguments for the alternative of the choice contexts and explanations for the contexts to be provided for the user during process execution (see Figure 15).

Complex process fragments composed of several substeps are modeled as plan contexts. The control flow of a plan context is defined using a language chosen in step 1.

Step 3 - Define Tool Capabilities in the Tool Model: The tool model serves as high-level specification of the tool capabilities.

In case of developing new tools the identification of the required tool categories is mainly determined by the structure of the underlying product model. For each sub product model (document type) a tool category is defined like an ER editor for ER diagrams. The tool category is then related to the products and to all actions operating on these products in the process model. For example, all

actions dealing with ER diagrams or its components would be related to the tool category *ER editor*, whereas all actions operating on data flow elements would be assigned to a *DFD editor*. In addition, for each tool category the shapes used to display the products, the display association between the products and the shapes, and the command elements to be provided must be defined.

For modeling a legacy tool, the capabilities accessible through its APIs are represented in the tool model. The legacy tool is defined as *tool category* and the actions (and their input parameters) accessible through the service invocation API are defined as action types including the output parameters accessible through the feedback API. Moreover, the command elements accessible through the command element API and the product representations accessible through the product display API are defined. All those definitions are related to the tool category of the legacy tool.

Step 4 - Define the Environment Model: Once the relation of a tool category to its products and actions has been established in the tool model, the tool category can be made responsible for a set of executable by instantiating the *provides_ executable_context* association. In addition, the choice contexts to be executed by the tool category are related to the tool category via the association *provides_choice_context*. Each alternative of an associated choice context must be related to a command element of this tool category (via the association *display_of_intention*) to define how the intention of each alternative should be displayed (activated).

The process of modeling the tool capabilities (step 3) and then relating them to the executable and choice contexts iterates until all executable and choice contexts are associated with a tool category.

Step 5 - Implement Domain-Specific Tool Functionality and/or Wrappers for Legacy Tools: Depending on whether a new tool is being built or an existing legacy is being wrapped the implementation tasks differ.

In both cases the tool implementation framework significantly facilitates the implementation effort. It relieves the tool builder from taking care about the general control flow, the message exchange with the enactment domain, the context execution and the context detection. The generic mechanisms ensure that whenever an executable context shall be executed the associated action is invoked, and that whenever a choice context shall be executed the user interface is adapted according to the choice context specification, and that the user interactions are matched against the context definitions of the environment model.

The components beyond the generic implementation framework are the domain specific tool actions, the specific user interface classes for displaying products using specific shapes, and the repository layer for storing and retrieving tool specific product data.

For new tools the domain-specific tool components have to be implemented. The *implementation* of a typical action requires about one to two pages of C++ code. The effort for implementing a new product shape or control icon heavily depends on the user interface toolkit used, but normally it does not exceed two pages of code.

For legacy tools wrappers for the domain-specific tool functionality have to be implemented by overwriting specific adapter classes of the implementation framework. According to our experience, *wrapping* a legacy tool action or user interface element requires approximately five to ten times less code than implementing it from scratch (depending on the API provided by the legacy tool).

Step 6 - Integrate Required Enactment Mechanisms: For each plan context definition language chosen in step 1, an enactment mechanism (interpreter) has to be embedded in the generic process engine framework. This is achieved by specializing the abstract base class representing the `Deduce-Context` state (see Section 5.2). The amount of work required depends on whether an enactment mechanism is available and whether it provides the required interfaces. If there exists an enactment mechanism providing the required interfaces, the effort required for the integration is marginal.

8.2 An Example: Building the PRIME-CREWS Environment

We illustrate the six steps by describing the implementation of the requirements engineering environment PRIME-CREWS. All tools of the PRIME-CREWS environment were built from scratch.

Step 1 - Choose One or More Plan Context Definition Languages: We first experimented with the Petri-Net language SLANG. It was fairly easy to represent the concepts of the contextual process meta model in SLANG (see Section 4.1.3). By applying SLANG to define the method guidance (step 2) it turned out that defining complex control flows like branches and loops was not always straightforward. We had to introduce many additional control transitions and places which merely served for emulating common control constructs like loops and branches. We therefore used the imperative language C++ for defining such contexts since C++ offers more suitable constructs.

Step 2 - Definition of Method Guidance in the Process Model: The product model underlying the PRIME-CREWS environment is structured according to the three dimensions of requirements engineering [77]. It adds a conceptual goal model, and a model for structuring multi media artifacts such as real world scenes to the product models provided by its pre-cursor environment PRO-ART (ER model, data flow model, hypertext model, gIBIS-like decision model, the RSM-model; see [4], [78] for details). Moreover, PRIME-CREWS extends the dependency model of PRO-ART by adding link types for interrelating the scenes and the conceptual models. On top of these product models 159 situation types have been defined. By relating these situation types to the defined intentions, 245 executable contexts, 38 choice contexts, and 82 plan contexts have been defined. The initially small number of choice contexts and plan contexts progressively increased due to the acquisition of method knowledge during our trial applications (see Section 9.4 for an example).

Step 3 - Definition of the Tool Capabilities in the Tool Model: For each of the eight product sub models we defined a tool category, namely the ER editor, DFD editor, hypertext editor, decision editor, dependency editor, RSM editor, goal editor, and the whiteboard editor. In addition, three product independent tools were defined: the model browser providing an overview on the product models, the task manager for managing pending tasks, and the topic manager for collecting and structuring open topics. Moreover, the actions provided by each tool category have been defined. Altogether we defined 245 elementary actions.

In addition we have predefined a set of common command elements such as the pull-down menus *Document, Edit, Tools, Preferences, Help* and specific icons for opening a model, adding a model element etc. These command elements are used by all tools. Thereby we ensured that an intention provided in more than one tool is activated by the same command elements. Moreover, we defined a set of generic shapes which can be used in all tools such as rectangular, circles, oval boxes, triangles and uni/bidirectional arrows.

Step 4 - Definition of the Environment Model: In the environment model, the actions of the eleven PRIME-CREWS tool categories were related to the 245 executable contexts defined in the process model. Moreover, the choice contexts were related to the various tool categories, including the assignment of at least one command element to each alternative of choice context.

Step 5 - Implementation of Domain-Specific Tool Functionality: All tools have been implemented using the generic tool architecture of the PRIME implementation framework. For each tool category the associated actions have been implemented by specializing the corresponding classes of the generic framework. Similarly, command elements, special shapes and the product layer have been implemented. Through the use of the generic framework all the tools have the same structure and they share about 60-70 percent of their code.

Step 6 - Integration of the Required Enactment Mechanisms: We have realized enactment mechanisms for C++ and SLANG. In the case of C++, the plan contexts were specified as subclasses of the abstract class `CplusplusPlanContext`. Each of these subclasses specializes the `deduceNextContext` method. This specialization defines the control flow of the corresponding plan context. The specialized method is invoked by the generic process engine architecture during the enactment of the plan context. For enacting plan contexts defined in SLANG we have implemented a limited SLANG interpreter and embedded it into the generic process engine architecture. In both cases the enactment mechanisms could be easily plugged into the generic process engine architecture.

8.3 Lessons Learned

Building a domain specific PIE comprises four modeling and two implementation activities. Whereas the modeling activities are typically performed by the method engineer, the

implementation activities are executed by the tool builder. We summarize our experience gained from building the PRIME-CREWS and TECHMOD environments.

8.3.1 Method Engineer: Defining and Adapting Process, Tool and Environment Models

It turned out that defining processes using the three context types offers some significant advantages to the method engineer in comparison with just applying, e.g., a Petri-Net based process modeling language. The three context types provide a guideline for the method engineer on how to structure process models, regardless of the specific enactment language chosen. In addition, and in contrast to other process modeling approaches, the method engineer is forced to make decision points explicit (by defining choice contexts).

The explicit definition of the tool capabilities forces the method engineer to think about the "right" granularity of tool functions defined as executable contexts. Moreover, the method engineer is aware of the existing tool support.

The integration of services provided by the tools and the services defined in the process model can be easily achieved. Moreover, the explicit definition of the tool capabilities empowers the method engineer to consider the available tool support during the method definition. The explicit tool definitions support her or him in defining method fragments with the right granularity.

Most importantly, the explicit definition of processes and tool capabilities enables an easy adaptation of the guidance offered by the environment. This is essential if the support offered has to be adjusted to project-specific needs and in (experimental) settings, in which new knowledge about good process performance is constantly elicited and learned.

8.3.2 Tool Builder: Implementing Domain-Specific Process-Integrated Tools

The implementation of 16 process-integrated tools for the PRIME-CREWS and TECHMOD environments confirmed our assumption that the concepts of the tool model and their interrelation with the concepts of executable and choice context are sufficient for defining tool services. There was no single situation in which a tool service could not be adequately described using executable or choice contexts. Moreover, the developers were forced to define process knowledge explicitly in plan contexts instead of embedding it in the code of the tools. In other words, the "process in the tool syndrome" [34] was avoided.

The implementation of the tools was significantly facilitated by the generic tool architecture and the reuse of the generic implementation framework. It turned out that the generic architecture enables the programmer to extend the tool functionality without investing time in understanding the

structure of the tool or having it implemented. For example, the implementation of an action (executable context) was possible without being aware of the other tool actions and/or worrying about the control flow of the program.

Moreover, the predefined "slots" of the framework and their interfaces enforced the programmers to produce modular code. Thus tool maintenance was eased. Each tool could be easily maintained and extended by any programmer. In addition, the architecture improved the communication between the programmers and served us as means for distributing the work between them.

Of course, the programmers have to be trained in implementing process-integrated tools using the PRIME implementation framework. This requires about 2 weeks in average.

In comparison with the development of earlier prototypes we observed a time reduction by at least a factor of two for implementing a tool like the DFD editor (for PRIME-CREWS) or the FlowSheet editor (for TECHMOD), although the programmers of the new tools had not implemented the previous ones.

8.4 A Sample Session with PRIME-CREWS

In the following we highlight the benefits a PRIME-based PIE offers for the application engineers. We describe the enactment of the plan context *PC_SubtypeEntityAndAdjustDFD* defined in Section 4.1.3 in the PRIME-CREWS environment to illustrate how a PRIME-based environment

- Supports the application engineer in activating method fragments (Section 8.4.1);
- Explains new method fragments to the application engineer (Section 8.4.2);
- Executes automated process steps (executable contexts; Section 8.4.3);
- Provides methodical advice during choice context execution (Section 8.4.4);
- Supports the engineer in providing correct feedback information (Section 8.4.5).

As background for the example we assume that the PRIME-CREWS environment is used for modeling a library system. Anita, the application engineer, is receiving method guidance for subtyping the entity type *publication* in the current ER diagram into *books* which can be checked out for four weeks and *journals* which cannot be checked out.

In the following, the paragraphs written in normal font describe the application engineer's interactions, while the paragraphs written in italics give an explanation of what happens at the technical level of the PRIME framework[10].

---

[10] The description of the user interactions and the explanation of what happens at the technical level take a much longer time to read than it would take to experience the advantages yourself by using the PRIME-CREWS environment.

8.4.1 Tool Supported Invocation of Method Fragments

Anita selects the entity type *publication* and the *Subtype Entity* menu item in the ER editor (see Figure 14). Since the interactions match with a context definition, Anita has activated the execution of a context, in this case of the plan context *PC_SubtypeEntityAndAdjustDFD*. Note that she needs not to be aware about the plan context definition, since a plan context is activated in a process-integrated tool just like any other tool functionality (context).

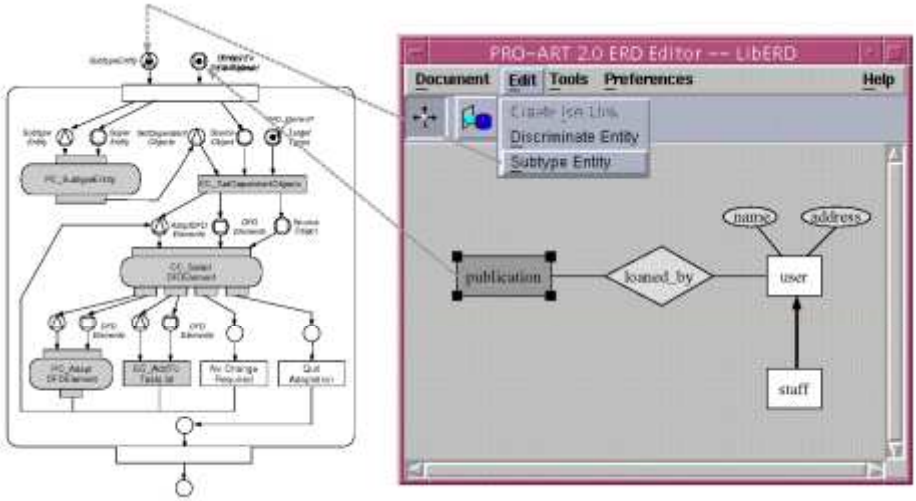

Figure 14: Activation of the plan context PC_SubtypeEntityAndAdjustDFD in the ER editor.

The ContextMatcher of the ER editor matches the selected entity and the activated menu item with the situation and intention of the context definitions. It identifies the activation of the plan context PC_SubtypeEntityAndAdjustDFD and sends an enactment request to the enactment domain with the context identifier and the situation instance (the selected entity publication) as parameter. The process engine becomes active, loads the definition of the plan context PC_SubtypeEntityAndAdjustDFD from the process repository, and initializes the input places of the corresponding SLANG net with the parameters received with the enactment request. For example, the situation place Entity_To_Be_Subtyped is filled with a data token representing the entity type publication (illustrated in Figure 14).

8.4.2 Method Advice: Explanations of Method Fragments

As a result of the plan context invocation, PRIME-CREWS displays an explanation of the activated plan context to Anita. The textual description explains the overall goal of the method fragment and the sequence of steps to be performed. Anita can now decide whether the method advice should be displayed the next time the context is invoked or not.

For each executable, choice, or plan context the method engineer should provide a textual description. This description is retrieved when the method fragment is invoked and displayed in the method advisor window. By clicking on the name of the sub context (in the case of a plan context) or alternative context (in the case of choice context) the user can access more detailed information for each step in a hypertext-like manner. The display of the method advice for a new defined context is enabled by default. It can be disabled by the user for further invocations after it has been displayed once.

Anita acknowledges the information obtained from the method advice window and presses the disable button. Thereby she states that whenever this context is activated again, the textual description should not be displayed automatically. Moreover, by pressing the *Start Context* button she activates the enactment of the plan context (she could also have stopped the enactment of the plan context).

According to the plan context specification, first the sub plan context PC_SubtypeEntity is activated.

The plan context *PC_SubtypeEntity* guides Anita during the specialization of the entity type. She creates two sub entities, namely the entity `book` and the entity `journal`, for the entity `publication` (not explained in detail here). In addition, the sub-entities are automatically related to the super-entity via *IsA-links*.

8.4.3 Automated Process Performance

According to the plan context definition, an automated process step (executable context) is performed after the plan context *PC_SubtypeEntity* is finished. This automated process step retrieves all DFD elements from the repository which might be effected by the subtyping of the entity. It thus represents a traceability strategy which ensures the use of recorded trace information (in this case the dependencies between the entity and the DFD elements).

By performing the executable context EC_GetDependentObjects provided by the dependency editor all data flow elements which are related to the entity publication via a dependency link are retrieved from the repository[11]. Among the objects returned are the data store PUBLICATION and its adjacent data flows. They build the situation of the choice context CC_SelectDFDElement which is deduced as the next context to be executed by the process engine and passed to the dependency editor, who is, according to the environment model, responsible for executing this context.

---

[11] PRIME-CREWS's mechanisms and models for supporting traceability between specification objects (such as entity types and data stores) via a comprehensive dependency structure described in [4] in detail.

8.4.4 Method Advice: Supporting User Choices

The dependency editor adapts its user interface according to the definition of the *CC_SelectDFDElement* context and the actual situation data. It displays the four intentions of the alternatives as defined in the environment model and displays the tokens of the situation place, namely the retrieved DFD objects and the entity proceedings.

To get more information on the four alternatives, Anita requests additional guidance from the dependency editor by selecting the guidance menu item.

The arguments for choosing an alternative are retrieved from the process repository and displayed in the method advice window (Figure 15). Again, it is possible to browse through the alternatives.

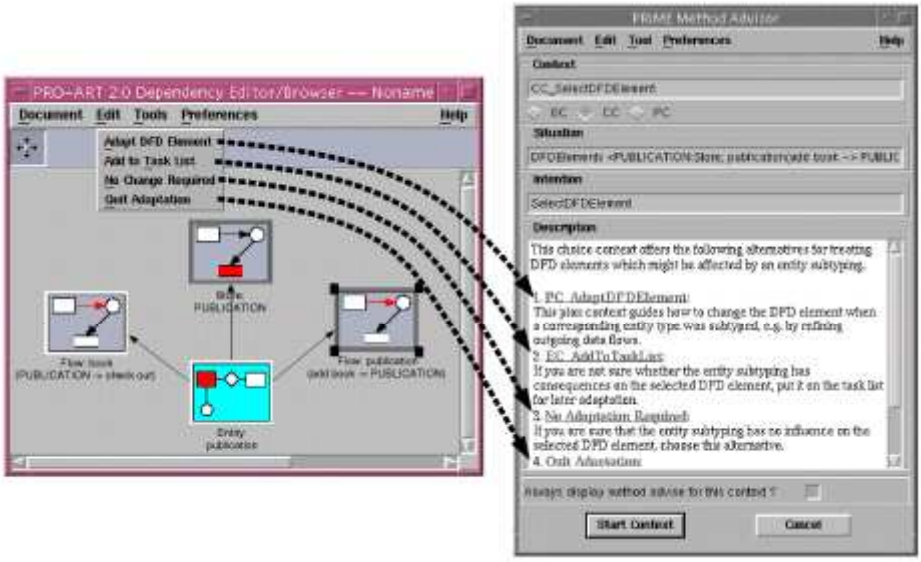

Figure 15: Dependency editor performing the choice context CC_SelectDFDElement.

Anita looks at the arguments for the four alternatives and decides to adapt the flow publication defined between the store `PUBLICATION` and the process bubble *Check Out*. By selecting the shape representing the flow in the dependency editor and choosing the menu item *AdaptDFDElement* from the *Edit* menu, Anita initiates the execution of the plan context *PC_AdaptDFDElement*. This complex plan context guides Anita in the adaptation of the selected data flow (not explained here).

After the adaptation is finished, the choice context *CC_SelectDFDElement* is again activated in the dependency editor. Now only the store PUBLICATION and flow publication from process book to store PUBLICATION can be selected. These products are highlighted by the tool using dark borders (Figure 15).

Due to the control flow of the plan context, the token which represents the adapted flow has been consumed by the execution of the choice context described above. In the dependency editor

only product parts which are part of the actual situation (for which a token exists in the DFD_Element place of the plan context) are displayed as selectable.

As the next item to be adapted Anita selects the flow *publication* (between the process bubble *add_book* and the store `PUBLICATION`) and the menu option *AddtoTaskList* since she is currently not sure whether this element is actually effected.

This interaction matches with the executable context EC_AddToTaskList which is executed by the task manager (brokered via the process engine).

Finally, she finishes the plan context enactment by choosing the alternative *Quit_DFD_Adaptation*.

8.4.5 Assuring Correct Feedback Information

The adaptation of the user interface during the execution of the choice context *CC_SelectDFDElement* ensures that the items selected by Anita are understood by the process engine. It guarantees that an appropriate reaction to the feedback is defined by the process fragment. Since the abortion of the adaptation of the DFD elements is offered as one alternative, Anita can deviate from the guidance provided by the plan context execution whenever she wants, but in a defined way.

8.4.6 Feedback Obtained from Users

In our early validation studies with the PRIME-CREWS and TECHMOD environments, most users reported that the reflection of the actual method definitions and the current enactment state in the behavior of the tools provides very helpful guidance; especially the adaptation of the user interface according to the method definitions and the support offered for invoking predefined method fragments. Since attention is automatically drawn to the applicable product parts and services, wrong and unintended interactions are avoided. The uniform activation of tool services (executable and choice contexts) as well as process fragments (plan contexts) was regarded as a significant improvement.

In contrast to our previous prototypes (and most other process-centered environments) the number of different interfaces the user has to cope with was reduced. Now the user essentially interacts with the enactment domain via the normal development tools (not through isolated guidance interfaces). This was also reported as a major improvement.

9 Change Integration in a PRIME-Based Environment

Method guidance continuously evolves due to various reasons such as changes in organizational and/or project policies, increase and/or revision of method knowledge, adding and/or modification of tool functionality. The process of adapting a process integrated environment (PIE) to such changes is thus as (or even more) important as building a PIE from scratch.

One major drawback of current process-centered engineering environments is that method guidance is hard-coded in the tools. The adaptation to changes is thus difficult and labor-intensive, if not impossible. Embedding a new tool in an environment is generally difficult since the method guidance encoded in the tool interferes in most cases with the method definitions. Moreover, since method guidance, tool capabilities and their interrelations are not explicitly represented, it is hard to determine how a method change affects the tools and vice versa.

In contrast, a PRIME-based PIE facilitates the incorporation of a change in two major ways.

First, the conceptual modeling of processes, tools, and their integration in the environment model empowers a change definition on the modeling level. Once a change is defined at the modeling level, the effect of the change can be analyzed. Moreover, the environment meta model supports the selective retrieval and thereby the reuse of method fragments (contexts of any type) and tool functionality and thereby eases the change implementation.

Second, the interpretation of the environment model by all PRIME components ensures that changes can be mainly accomplished at the modeling level. The integration of most changes requires only an adaptation of the method, tool, and/or environment models. If changes at the implementation level are required, for example, if a new elementary process step shall be implemented, the implementation is supported by the generic components of the PRIME implementation framework.

For integrating a change into a PRIME-based environment, we propose a five step strategy:

(Step 1) *Change definition in the environment model*: The environment model has to be adapted according to a change request. In other words, the change is first defined at the modeling level;

(Step 2) *Analysis of change affects*: The meta models are used to analyze the existing definitions in the current environment model and to retrieve those parts of the definition which are effected by the changes made in step 1;

(Step 3) *Adaptation of effected tool and/or method definitions*: The effected tool and/or method definitions have to be adapted in a way that the tool and method definition are consistent;

(Step 4) *Establishing a consistent environment model*: Newly defined or changed method fragments and/or tool capabilities must be interrelated to achieve a consistent environment model definition;

(Step 5) *Implementation of missing tool functionality*: In some cases, a change cannot be fully implemented through model adaptation and thus requires, in addition, the implementation and/or wrapping of tool functionality.

The strategy outlined above can also be applied for transferring method guidance and tool functionality between different application domains. As our experience with the development of the TECHMOD environment indicates, many method fragments and tool functionality developed for the requirements engineering environment PRIME-CREWS could be reused for the TECHMOD environment. Most of the required adaptations could be achieved through model changes.

In the next sections we elaborate on the integration of a change request in a PRIME based environment. Therefore, we classify change requests into three main categories, namely method change requests (Section 9.1), tool change requests (Section 9.2), and responsibility change requests (Section 9.3).

9.1 Integrating Method Change Requests

According to our experience, the adaptation of the method definitions is the most common change request. Such changes are typically raised by the need to adjust the support offered to project-specific needs, to implement new methodical knowledge or to enrich existing method fragments.

Step 1 - Change Definition in the Environment Model: For accommodating method changes in a PRIME-based environment, the context definitions have to be adapted. According to our experience method changes seldom require the definition of completely new method fragments. Once a consolidated base of method fragments exists, method changes can be mainly achieved by adapting existing contexts (for example, by adding a new alternative to a choice context or by changing the control flow of a plan context) and/or composing existing contexts into an existing or new plan context. The environment meta model provides an excellent starting point for retrieving reusable method fragments. For example, if the method engineer defines the situation and/or intention of the new context, she or he can retrieve all contexts which are potential reuse candidates such as all contexts which are based on same or similar situation.

Step 2 - Analysis of Change Affects: Modifying existing and/or adding new executable and/or choice contexts can require an adaptation of the tool models. By applying the consistency constraints defined in Section 4.3 the concepts of the tool models which are effected by the method changes can be retrieved. For example, by applying constraint E1 and E2 it can be checked if a tool category is responsible for the action assigned to an executable context and if there is a mismatch between the input and output parameters defined for the action in the tool model and the situation defined for the action in the process model. If no tool category is responsible for executing the context, it can be checked whether a tool provides the required action or whether the definition of a new tool action is required. Similar, by applying constraints C1 and C2, it can be checked whether the tool responsible for the choice context is able to display

the intentions of all alternative contexts and whether all product types being part of a situation type of any alternative context can be displayed.

Step 3 - Adaptation of Effected Tool and/or Method Definitions: The conflicts between the tool definitions and the adapted method definitions detected in step 2 have to be resolved. This can either be achieved by adapting the tool definitions or by changing the method definitions. In the case of an executable context, for example, if the required action is not provided by any tool, the method engineer can either define a new action and assign it to a tool category, or revise the definition of the executable context. For example she or he can define a new plan context which achieves the method guidance by composing more fine-granular actions provided by the tools. Similarly, if there is a mismatch between the input and output definition of the actions either the input/output parameters in the tool model are adapted or the definition of the executable context is changed.

In the case of a new or changed choice context it might be the case that the tool which is or should be responsible for executing the context does not offer all required command elements and/or shapes. To solve this inconsistency, either the tool model is enhanced by defining the missing command elements and/or the shapes, or, the method engineer adapts the choice context definition instead of changing the tool model, or the adjustment is achieved by a combination of both.

Step 4 - Establishing a Consistent Environment Model: The newly defined or changed method fragments (contexts) and/or tool capabilities must be interrelated with each other. The definition of the associations and the support provided by the meta models is described in Section 8.1, step 4.

Step 5 - Implementation of Missing Tool Functionality: If in step 3 the tool definitions have been changed, the implementation must be adapted according to those changes. This means that either the wrappers of the legacy tools must be adapted, or that the missing actions, shapes, and command icons have to be implemented. As described in step 5 (Section 8.1) the PRIME implementation framework significantly facilitates those implementations. In contrast to the actions, shapes and command items, the menu entries and the short-key bindings are automatically generated by the PRIME implementation framework.

9.2 Integrating Tool Change Requests

Tool model modifications stem from enhancements or modifications of the available tool functionality initiated through, e.g., a user request for a new tool action, or a completely new tool, or the removal of an existing tool from the environment.

Step 1 - Change Definition in the Environment Model: Changes in the tool implementations must be reflected at the modeling level by adapting the tool model accordingly. Ideally, changes are first defined in the tool model and then, after establishing a consistent environment model, implemented. For example, if a new tool action shall be implemented the action and the *input* and *output* parameters should be defined in the tool model. Similarly, if an existing tool shall be

replaced by a new tool, the capabilities of the new tool should be defined in the tool model. The environment model can then be used to detect and analyze the differences in the capabilities provided by the old and the new tools, for example, to detect capabilities provided by the old tool but not by the new one.

Step 2 - Analysis of Change Affects: Changes of the tool capabilities do normally not affect the method fragment definitions. However, if a capability required for executing an executable and/or choice context is changed, the method definitions are effected.

For example, if the signature of an action has been modified, it has to be assured (by checking constraint E1 and E2) that the new definition still fits with the situation defined for the action in the method definitions. In the case of a legacy tool, it has in addition to be checked if the wrappers still work as desired. Otherwise the association of the tool category with the corresponding executable context becomes invalid. Or, a modification of command elements and/or the visualization of products may interfere with the associations defined between the tool categories and the choice contexts, for example, if a command element has been removed or changed, it must be checked whether and which alternatives of a choice context are effected.

Step 3 - Adaptation of Effected Tool and/or Method Definitions: The modification of the tool capabilities can cause certain adaptations of the method fragments and the associations defined between the fragments and the tool capabilities. Similar to the integration of method changes, the adaptation can be achieved by modifying the tool and/or method definitions. For example, if a certain tool capability like an action or a shape has been removed the associated contexts have to be changed or new capabilities have to be defined in the tool model. Similar, to enable the execution of new actions defined in the tool model corresponding executable contexts have to be defined in the process model.

Step 4 - Establishing a Consistent Environment Model: see Section 9.1, step 4 for a description.

Step 5 - Implementation of Missing Tool Functionality: If there are capabilities not provided by the tools, those capabilities must be implemented. If additional capabilities provided by a legacy tool shall be used, those capabilities must be wrapped as described in Section 7.

9.3 Integrating Responsibility Change Requests

Besides the method and tool changes, a change request can be concerned with assigning the responsibility for a context execution to another tool.

Step 1 - Definition of Change in the Environment Model: A change in the responsibility for executing an executable or choice context can be achieved by adapting the associations defined between the tool and process models.

Step 2 - Analysis of Change Affects: Before changing an assignment of an executable or choice context, the consequences of this change should be explored.

For example, if instead of a tool category T_1 a tool category T_2 shall be responsible for executing an executable context, it should be checked if T_2 actually provides the action related to the executable context and if the definition of the input and output parameters corresponds to the situation of the executable context. This can be achieved by checking the constraints E1 and E2 (Section 4.3). Similarly, when assigning a choice context to another tool category, the method engineer can check by using the constraints C1 and C2 if the new tool category provides all command elements and product shapes required for displaying the alternatives of the choice context to be re-assigned.

The steps three, four and five have only to be performed if the intended change requires the definition of new tool functionality or the adaptation in the method definition. These cases are described in Section 9.1 and Section 9.2 respectively.

9.4 Adapting Method Guidance: A Small Example

We illustrate the ability of PRIME-based environments to adapt to changes using an example of the PRIME-CREWS environment. The example illustrates the adaptation of the method guidance for specializing an entity and adjusting the effected data flow diagrams. According to our experience, adapting the method guidance is by far the most frequent type of change in a PIE.

The original method guidance for the specialization of an entity was defined in a plan context *PC_SubtypeEntity* which supports the creation of sub-entities and the *IsA-Links* between the super-entity and its sub-entities.

A review of the requirements specifications produced with the PRIME-CREWS environment reveals that specialization in the entity relationship diagrams are often not accurately reflected in the corresponding data flow diagrams. Therefore, the method engineer defined a new plan context *PC_SubtypeEntityAndAdjustDFD* which guides the requirements engineer in the specialization of the entity and the adjustment of the data flow diagrams. We described this plan context already in Section 4.1.3 (see also Figure 4).

In the following we briefly sketch how the definition of the new plan context was supported by the modeling tools and achieved in the PRIME-CREWS environment.

Step 1 - Change Definition in the Environment Model: For defining the new plan context, the method engineer (Fritz) first queried the environment model to obtain all contexts which deal with the modeling of entities and the adaptation of data-flow diagram elements. From the 21 retrieved contexts, Fritz selected four to be reused:

- The plan context *PC_SubtypeEntity* which guides the specialization of an entity;

- The plan context *PC_AdaptDFDElement* which guides the adaptation of a single data-flow diagram element;
- The executable context *EC_GetDependentObjects* which selects objects with the specified association from the repository;
- The executable context *EC_AddToTaskList* by which an open topic is added to the workplace specific task list;

Fritz reused these contexts for defining the new method guidance in a plan context. He first defined the situation and the intention of a new choice context (*CC_SelectDFDElement*) which allows the requirements engineer to select the dependent DFD object and choose an adaptation strategy. He then associated the alternatives *PC_AdaptDFDElement*, *EC_AddToTaskList*, *NoChangeRequired*, *QuitAdaptation* to the new choice context. According to the environment model definitions, the alternatives *PC_AdaptDFDElement*, *EC_AddToTaskList* are executed by different tools (the TaskManager and the DFD editor). He then embedded the choice context into the plan context definition and completed the plan context definition by specifying the control flows and by embedding additional contexts as shown in Figure 4.

As illustrated by the example, the definition of a new plan context is often an activity by which existing contexts are "glued" together.

Step 2 - Analysis of Change Affects: After finishing the method definitions, Fritz checked the constraints on the current model definitions. As a result he noticed that the new choice context *CC_SelectDFDElement* is not related to any tool category.

Step 3 - Adaptation of Effected Tool and/or Method Definitions: In our example, there is no need to adapt the tool models, since all required action and command elements are provided by the defined tools.

Step 4 - Establishing a Consistent Environment Model: Fritz thus just had to relate the choice context *CC_SelectDFDElement* to the tool category which should be responsible for executing the context, namely the dependency editor. This assignment required in addition, that the intentions of the four alternative contexts are related to command elements of the DFD editor (see Section 4.3.2 for details). Thereby the DFD editor was made responsible for executing the choice context.

Step 5 - Implementation of Missing Tool Functionality: The definition of the new method guidance in our example did not require any changes at the implementation level. Even the new tool interoperations required by the changes were achieved without any single source code change.

As our experience indicates, changing method guidance in a PRIME-based environment requires much less effort than adapting the method guidance in existing process-centered environments or in conventional CASE tools.

10 Conclusions and Outlook

We presented PRIME, a framework for *PR*ocess-*I*ntegrated *M*odelling *E*nvironments. The development of the PRIME framework was driven by the requirements for process-integrated environments elaborated in Section 2. PRIME differs significantly from current PCEs in that it provides method guidance through process-integrated tools which are able to adapt their behavior to the method definition and to the actual process situation. Thereby the tools offer situated guidance to the humans performing the process. PRIME enables the user to initiate the enactment of method definitions and thereby empowers the user to play a more active role in process performance.

The main contributions of the PRIME framework (in comparison with existing PCEs) can be summarized as follows:

Integrated tool and process models (Section **4**): We argued that tools should not longer be treated as second class citizens and suggested to explicitly define the tool capabilities in addition to the process models. Moreover, we proposed to integrate both types of models forming the so-called environment model. The interpretation of the environment model by the components of the PRIME framework and the *integration of enactment and performance domains* (Section **5**) build the conceptual foundations for the process-integration of the interactive engineering tools. Process-integrated tools offer integrated, definition-conform method guidance and thereby significantly improve the consideration of the project-specific definitions by the stakeholders executing the process.

Implementation framework for process-integrated tools (Section **6**): The implementation of the generic tool architecture offers well-defined interfaces for embedding tool specific actions, shapes and command elements. It thus defines a quasi standard way of implementing process-integrated tools. The tool architecture ensures a synchronization with the enactment domain according to the interaction protocol definitions. Moreover, it facilitates the realization of project-specific changes and the implementation of a new tool through the automated adaptation of the tool behavior to changes in the environment model. This is mainly achieved through the ContextMatcher which supports the user in the unified activation of contexts of any type and the ContextExecutor which ensures that the contexts are executed as defined. In the case of an executable context, it performs the associated action; in the case of a choice context it restricts the selectable products, command elements and menu options according to the choice context definition and the current enactment state. Both components ensure that changes in the environment model are automatically reflected in the tool behavior.

Implementation framework for process engines (Section **6**): The language used for defining plan contexts depends on the type of support provided. For example, state-based languages are well suited to define simple advice patterns, whereas the effective definition of more elaborated support requires, to our experiences, the use of languages with a higher expressiveness such as Petri-Nets

or (visual) programming languages. Improving the method guidance could thus, at certain stages, require the use of an additional plan context specification language and thus the integration of a suitable interpreter in the PRIME-based environment. The integration of a new interpreter is facilitated by the interfaces offered by the generic enactment architecture which ensures interaction protocol conform synchronization with the performance domain.

Process-integration of legacy tools (Section 7): One cannot assume that for each required functionality new process-integrated tools are implemented. Achieving a process-integration of legacy tools the users are familiar with is thus essential. The PRIME framework significantly facilitates the process-integration of legacy tools by wrapping the tool using appropriate user interface and action wrappers into the generic tool architecture. Thereby a legacy tool is empowered to offer integrated, definition-conform stakeholder guidance.

Change integration support (Section 9): Easy change realization and appropriate change integration support are two important prerequisites for adapting requirements management environments to project-specific needs. Change integration is facilitated by the PRIME implementation framework in two major ways. First, model-based change definition and change analysis are empowered through the conceptual modeling of processes, tools, and their integration in the environment model. Second, the effort required for a change realization is significantly reduced. The interpretation of the environment model by the PRIME components ensures that the change implementation can mainly be achieved through model adaptations. If the integration of a change affects the implementation level, the realization is significantly supported by the generic tool and enactment architectures.

The generic components of PRIME have been implemented as reusable object-oriented implementation framework (approx. 120,000 lines of C++ code). In addition, modeling tools for supporting the definition of the tool, process and environment models have been implemented. The PRIME implementation framework and the associated development and customization strategies, were validated by implementing two prototypical process-integrated environments, PRIME-CREWS and TECHMOD, consisting of 16 process-integrated tools (approx. 240,000 lines of tool specific C++ code). Moreover, the PRIME framework was validated by applying the environments in small case studies and trial applications.

Most users of the PRIME-CREWS and the TECHMOD environments reported that the reflection of the actual method definitions and the current enactment state in the behavior of the tools provides very helpful guidance. The user is reminded of the project-specific definitions and, as a consequence, the definitions are much better considered during process execution. The uniform activation of tool services (executable and choice contexts) as well as method fragments (plan contexts) was regarded as a significant improvement. Project-specific guidance can thus be invoked by the user without even knowing their existence. Moreover, in comparison with our previous prototypes (and most other process-centered environments) the number of different user interfaces was reduced. Now the user essentially interacts with the enactment domain via the normal

development tools (not through isolated guidance interfaces). This was also reported as major improvement.

We are currently extending the PRIME implementation framework to enable context detection and invocation across tool boundaries. Moreover, to further reduce the implementation effort required for building a process-integrated tool we enrich the PRIME framework with tool generation approaches which enable the generation of tool specific functionality based on richer tool specifications.

In addition, we are investigating in the application of the PRIME framework for the implementation of *project-specific trace capture strategies* (see [22], [79] for details), and in the development a *comprehensive framework for the process integration of legacy tools*. The framework will define detailed criteria for determining the degree of process integration which can be achieved for a given legacy tool. The process-integration of a legacy tool will be facilitated by generic wrappers which can be adjusted according to the assessed capabilities of the tool. To empower as much as possible reuse, the wrappers will be related to the generic tool architecture of the PRIME implementation framework.

Acknowledgments: The authors like to thank their students S. Brandt, S. Ewald, M. Hoofe, T. Rötschke, K. Schreck, A. Spiegel, and W. Thyen. Without their enthusiasm, the implementation of the generic architecture and the PRIME-CREWS and TECHMOD environments would not have been possible.

References

[1] M. Paulk, B. Curtis, M. Chrissis and C. Weber, *Capability Maturity Model for Software: Version 1.1,* Technical Report SEI93-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, February 1993

[2] W. E. Deming, *Out of the Crisis*, Massachusetts Institute of Technology, Center for Advanced Engineering Study, Cambridge, 1986

[3] M. Dowson, *Consistency Maintenance in Process Sensitive Environments,* In: Proc. Process Sensitive Software Eng. Environments Architectures Workshop, Boulder, Colorado, USA, September, 1992

[4] K. Pohl, *Process Centered Requirements Engineering,* RSP marketed by J. Wiley & Sons Ltd., UK, 1996

[5] J. Lonchamp, *Software Process Modelling and Technology, Chapter An Assessment Exercise*, In: A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), Software Process Modelling and Technology, Student Research Press, Wiley & Sons, England, 1994, pp. 335-356

[6] V. Ambriola, G. A. Cignoni and C. Montangero, *The Oikos Services for Object Management in the Software Process*, In: B. Warboys (Ed.), Proc. of the Third Europ. Workshop on Software Process Technology, Villard de Lans, France, February, Springer-Verlag, LNCS, 1994, pp. 2-13

[7] S. Bandinelli, A. Fuggetta, C. Ghezzi and L. Lavazza, *SPADE: An Environment for Software Process Analysis, Design, and Enactment,* In: A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), Software Process Modelling and Technology, RSP, London, 1994, pp. 223-248

[8] N. Barghouti, *Supporting Cooperation in the MARVEL Process-Centered Software Development Environment,* In: Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, New York, New York, USA, 1992, pp. 21-31.

[9] N. Belkhatir, J. Estublier and W. L. Melo, TEMPO: *Enhancing O.O. Paradigm for Modeling Software Engineering Processes.*In: W. Schäfer (Ed.), Proc. of the Eighth Intl. Software Process Workshop: State of the Practice in Process Technology, Wadern, Germany, March 1993, pp. 37-39

[10] P. Boveroux, G. Canals, J.-C. Derniame, C. Godart, P. Jamart and J. Lonchamp, *Software Process Modelling in the ALF System: an Example,* In: V. Ambriola, A. Fuggetta and R. Conradi (Eds.), Proc. of the First Europ. Workshop on Software Process Modeling Technology, Milan, Italy, CEFRIEL, AICA, Working Group on Software Engineering, May 1991, pp. 167-179

[11] M. Deiters and V. Gruhn, *The FUNSOFT Net Approach to Software Process Management*, Intl. Journal of Software Engineering and Knowledge Engineering, Vol. 4, No. 2, 1994

[12] C. Fernström, *PROCESS WEAVER: Adding Process Support to UNIX*, In: L. Osterweil (Ed.), Proc. of the Second Intl. Conf. on the Software Process, Berlin, Germany, IEEE Computer Society Press, February 1993, pp. 12-26

[13] P. Heimann, G. Joeris, C.-A. Krapp and B. Westfechtel, *DYNAMITE: Dynamic Task Nets for Software Process Management,* In: Proc. of the 18th Int. Conf. on Software Engineering, 1996, pp. 331-341

[14] T. Mochel, A. Oberweis and V. Sänger, *Income/star: The petri net simulation concepts,* Journal of Mathematical Modelling and Simulation in Systems Analysis, Vol. 13, 1993, pp. 21-36

[15] V. Ambriola, R. Conradi and A. Fuggetta, *Assessing Process-Centered Software Engineering Environments,* ACM Transaction of Software Engineering and Methodology, Vol 6, No 3, 1997, pp. 283-328

[16] P. Armenise, S. Bandinelli, C. Ghezzi and A. Morzenti, *A Survey and Assessment of Software Process Representation Formalisms,* International Journal of Software Engineering And Knowledge Engineering, Vol. 3, No. 3, 1993, pp. 410-426

[17] B. Curtis, M. Kellner and J. Over, *Process Modeling,* Communications of the ACM, Vol. 35, No. 9, 1992, pp. 75-90

[18] A. Finkelstein, J. Kramer, and B. Nuseibeh (Eds.), *Software Process Modelling and Technology*. Advanced Software Development Series, RSP marketed by J. Wiley & Sons Ltd., Taunton, England, 1994

[19] A. Fuggetta and C. Ghezzi, *State of the Art and Open Issues in Process-Centered Software Engineering Environment,.* Journal of Systems and Software Vol. 26, 1994, pp. 53-60

[20] F. Harmsen and M. Saeki, *Comparison of four Method Engineering Languages*, In: S. Brinkkemper, K. Lyytinen and R. Welke (Eds.), Method Engineering: Principles of construction and tool support -- Proc. of the IFIP TC8, WG8.1/8.2 Working Conference on Method Engineering, Atlanta, Georgia, USA, Chapman & Hall, London, England, August 1996, pp. 45-62

[21] J.-P. Tolvanen, M. Rossi and H. Liu, *Method Engineering: Current Research Directions and Implications for Further Research,* In: S. Brinkkemper, K. Lyytinen and R. Welke (Eds.), Method Engineering: Principles of construction and tool support -- Proc. of the IFIP TC8, WG8.1/8.2 Working Conf. on Method Engineering, Atlanta, Georgia, USA, Chapman & Hall, London., England, August 1996, pp. 296-317.

[22] K. Pohl, R. Dömges and M. Jarke, *Towards Method-Driven Trace Capture,* In: Proceedings of the 9th International Conference on Advanced Information Systems Engineering, CAiSE '97, Barcelona, Spain, June 1997, pp. 103-116

[23] G. Alonso, D. Agrawal, A. Abbadi and C. Mohan, *Functionality and Limitations of Current Workflow Managment Systems,* IEEE Expert, 1996

[24] M. Nagl (Ed.), *Building Tightly Integrated Software Development Environments: The IPSEN Approach,* LNCS 1170, Springer Verlag, 1996

[25] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh, *Inconsistency Handling in Multi-Perspective Specifications*, IEEE Transactions on Software Engineering, Vol. 20, No. 8, 1994, pp. 569 – 578

[26] R. Dömges, K. Pohl, M. Jarke, B. Lohmann and W. Marquardt, *PRO-ART/CE: An Environment for Managing the Evolution of Chemical Process Simulation Models,* Modelling and Simulation, Special Issue of ESM-96, Society for Computer Simulation International, pp. 1012-1017

[27] S. Arbaoui and F. Oquendo, *Managing Inconsistencies between Process Enactment and Process Performance States*, In: W. Schäfer (Ed.), Proc. of the Eighth Intl. Software Process Workshop: State of the Practice in Process Technology, Wadern, Germany, IEEE Computer Society Press, 1993, pp. 24-27

[28] M. Dowson and C. Fernström, *Towards Requirements for Enactment Mechanisms,* In: B. Warboys (Ed.), Proc. of the 3rd Europ. Workshop on Software Process Technology, LNCS, Villard de Lans, Frankreich, Springer-Verlag, No. 772, February 1994, pp. 90-106

[29] W. Emmerich, *Tool Construction for Process-Centred Software Development Environments based on Object Databases,* PhD thesis, University of Paderborn, Germany, 1995

[30] C. Fernström, *State Models and Protocols in Process Centered Environments*, In: W. Schäfer (Ed.), Proc. of the Eighth Intl. Software Process Workshop: State of the Practice in Process Technology, Wadern, Germany, IEEE Computer Society Press, 1993, pp. 72-77

[31] C. Fernström and L. Ohlsson, *Integration Needs in Process-Enacted Environments*, In: Proc. of the 1st Intl. Conf. on the Software Process, 1991, pp. 142-158

[32] M. A. Gisi and G. E. Kaiser, *Extending a Tool Integration Language*, In: M. Dowson (Ed.), Proc. of the First Intl. Conference on Software Process, Redondo Beach CA, IEEE Computer Society Press, October 1991, pp. 218-227.

[33] M.Jarke, *Strategies for integrating CASE environments.* IEEE Software, March 1992, pp. 54-61

[34] C. Montangero, *The Process in the Tool Syndrome: Is It Becoming Worse*? In: Proc. of the 9th Intl. Software Process Workshop, Arlie, Virginia, USA, IEEE Computer Society Press, October 1994, pp. 53-56.

[35] S. M. Sutton and M. H. Penedo, *Process Based Software Engineering Environments Architectures Session Report,* In: Proc. of the Seventh Intl. Software Process Workshop: Communication and Coordination in the Software Process, Yountville, CA, IEEE Computer Society Press, 1991, pp. 14-21

[36] G. Valetto and G. E. Kaiser, *Enveloping Sophisticated Tools into Process-Centered Environments*, Journal of Automated Software Engineering, Vol. 3, 1996, pp. 309-345

[37] I. Thomas and B. A. Nejmeh, *Definitions of Tool Integration for Environments*, IEEE Software Vol. 8, No. 2, 1992, pp. 29-35

[38] A. I. Wasserman, *Tool Integration in Software Engineering Environments,* In: F. Long (Ed.), Proc. of the Intl. Workshop on Software Engineering Environments, Berlin, Germany, Springer-Verlag, 1990, pp. 137-149

[39] A. Brown, A. Earl and J. McDermid, *Software Engineering Environments: Automated Support for Software Engineering.* McGraw-Hill, 1993

[40] M. Chen, and R. J. Norman, *A Framework for Integrated CASE,* IEEE Software, March 1992, pp. 18-22

[41] ECMA-NIST, *A Reference Model for Frameworks of Software Engineering Environments*, No. TR/55 Version 3. ECMA & NIST, 1993

[42] K. Pohl and K. Weidenhaupt, *A Contextual Approach for Process-Integrated Tools*, In: Proc. of the 6th Eurpoean Software Engineering Conference (ESEC) and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zürich, Switzerland, September, LNCS 1301, Springer Verlang, 1997, pp. 176-192.

[43] G. Boudier, F. Gallo, R. Minot and I. Thomas, *An Overview of PCTE and PCTE+,* In: J.-C. Derniame (Ed.), Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Environments, Boston, MA, November 1988, pp. 28-30

[44] L. Wakeman and J. Jowett, *PCTE -- The Standard for Open Repositories*, Prentice Hall, 1993

[45] EIA, *The CDIF 1994 Interim Standard – Overview*, No. EIA/IS-106, ISBN 0-7908-0012-8. Electronic Industries Association, 1994

[46] OMG, *XMI Specification,* Object Management Group, Inc., 1998, ftp://ftp.omg.org/pub/docs/ad/98-10-05)

[47] K. Pohl, R. Dömges and M. Jarke, *Decision Oriented Process Modelling,* In: Proc. of the 9th Intl. Software Process Workshop, Arlie, VA, IEEE Computer Society Press, October 1994, pp. 124-128

[48] N. S. Barghouti and B. Krishnamurthy, *An Open Environment for Process Modeling and Enactment,* In: W. Schäfer (Ed.), Proc. of the Eighth Intl. Software Process Workshop: State of the Practice in Process Technology, Wadern, Germany, IEEE Computer Society Press, 1993, pp. 33-36

[49] N. S. Barghouti and B. Krishnamurthy, *Using Event Contexts and Matching Constraints to Monitor Software Processes,* In: Proc. 17th Intl. Conf. on Software Engineering, Seattle, Washington, USA, May 1995, pp. 83-92

[50] S. Bandinelli, E. Di Nitto and A. Fuggetta, *Supporting Cooperation in the SPADE-1 Environment,* IEEE Transactions on Software Engineering Vol. 12, No. 12, 1996, pp. 841-865

[51] G. Junkermann, B. Peuschel, W. Schäfer and S. Wolf, *MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment,* In: A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), Software Process Modelling and Technology, RSP, London, 1994, pp. 103-130

[52] S. P. Reiss, *Connecting Tools Using Message Passing in the FIELD Environment,* IEEE Software Vol. 7, No. 4, July 1990, pp. 57-67

[53] M. Cagan, *The HP SoftBench Environment: An Architecture for a New Generation of Software Tools,* Hewlett-Packard Journal, Vol. 41, No. 3, June, 1990, pp. 36-47

[54] SunSoft *The ToolTalk Service* (White Paper), Technical report, SunSoft Inc., June 1991

[55] OMG, *CORBA: Architecture and Specification,* Object Management Group, Inc., 1995

[56] K. Brockschmidt, *Inside OLE*, Second Edition, Microsoft Press, Redmond WA, 1995

[57] M. Anderson and P. Griffiths, *The Nature of the Software Process Modelling Problem is Evolving,* In: Proc. of the 3rd European Workshop on Software Process Technology, EWSPT '94, LNCS 772, 1994, pp. 31-34

[58] GOODSTEP-Team, *The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes,* In: Proc. of the Asia-Pacific Software Engineering Conference, Tokyo, Japan, 1994, pp. 410-420

[59] S. Kelly, K. Lyytinen and M. Rossi, *MetaEdit+ -- A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment*, In: Proc. of the 8th Intl. Conference on Advanced Information Systems Engineering, LNCS 1080, Heraklion, Crete, Greece, 1996, pp. 1-21

[60] K. Lyytinen, P. Marttiin, J.-P. Tolvanen, M. Jarke, K. Pohl and K. Weidenhaupt, *Case Environment Adaptability: Bridging the Island of Automation*, Proc. of the 8th Annual Workshop on Information Technologies and Systems WITS '98, associated with the Intl. Conference on Information Systems (ICIS), University of Jyväskylä, Finland, Computer Science and Information System Reports TR-19, December, 1998

[61] G. Canals, N. Boudjlida, J.-C. Derniame, C. Godart and J. Lonchamp, *ALF: A Framework for Building Process-Centred Software Engineering Environments,* In: A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), Software Process Modelling and Technology, RSP, London, 1994, pp. 153-186

[62] R. Conradi, M. Hagaseth, J.-O. Larsen, M. Nguyen, B. Munch, P. Westby, W. Zhu, M. Jaccheri and C. Liu, *EPOS: Object- Oriented Cooperative Process Modelling*, In: A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), Software Process Modelling and Technology, RSP, London, 1994, pp. 33-70.

[63] D. Garlan and E. Ilias, *Low-cost, Adaptable Tool Integration Policies for Integrated Environments,* In: Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments, Vol. 15, 1990

[64] R. Orfali, D. Harkey and J. Edwards, *The Essential Distributed Objects Survival Guide,* John Wiley & Sons, 1996

[65] F. Griffel, *Componentware: Konzepte und Techniken eines Softwareparadigmas,* dpunkt Verlag Heidelberg, Germany, 1998 (in German)

[66] C. Rolland and G. Grosz, *A General Framework for Describing the Requirements Engineering Process,* In: Proc. of the Intl. Conf. on Systems, Man, and Cybernetics, San Antonio, Texas, USA, IEEE Computer Society Press, October 1994

[67] V. Plihon and C. Rolland, *Modelling Ways-of-working,* In Proc. of the 7th Intl. Conference on Advanced Information Systems Engineering (CAiSE '95), Jyväskylä, Finland, Springer Verlag, 1995, pp. 126-139

[68] C. Rolland, C. Souveyet and M. Moreno, *An Approach to Defining Ways of Working,* Information Systems, Vol. 20, No. 4, 1995, pp. 337-359

[69] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design,* Prentice Hall, 1991

[70] R. Klamma, *Prescriptive Process Definitions: Definition, Implementation, and Validation in Pro-ART,* Master's thesis, RWTH Aachen, Aachen, Germany, 1995 (in German)

[71] D. Harel, *STATECHARTS: A Visual Formalism for Complex Systems,* Science of Computer Programming Vol. 8, 1987, pp. 231-274

[72] M.Nagl and W. Marquardt, SFB-476 IMPROVE: Informatische Unterstützung übergreifender Entwicklungsprozuesse in der Verfahrenstechik, In; M. Jarke, K. Pasedach, K. Pohl, (Eds.) Informatik '97, Jahrestagung der Gesellschaft für Informatik, Aachen, Germany, Springer Verlag, 1997, pp. 143-154. (in German)

[73] M. Nagl and B. Westfechtel (Eds.), *Integration von Entwicklungssystemen in Ingenieuranwendungen,* Springer-Verlag, Berlin, Germany, 1998

[74] M. Jarke and W. Marquardt, *Design and Evaluation of Computer Aided Process Modelling Tools,* In: Intelligent Systems in Process Engineering, IPSE '95, Snowmass, USA, 1995

[75] R. Dömges, K. Pohl, and K. Schreck, *A Filter-Mechanism for Method-Driven Trace Capture,* In: Proc. 10th Int'l. Conf. Advanced Information Systems Engineering (CAiSE '98, Pisa, Italy, June 1998, pp. 237-250

[76] D. Garlan, R. Allan and J. Ockerbloom, *Architectural Mismatch or Why it's Hard to Build Systems out of Existing Parts,* In: Proc. 17th Intl. Conference on Software Engineering, Seattle, Washington, USA, 1995, pp. 179-185

[77] K. Pohl, *The Three Dimensions of Requirements Engineering: A Framework and its Application,* Information Systems, Vol. 19, No. 3, 1994, pp. 243-258

[78] P. Haumer, K. Pohl and K. Weidenhaupt, *Requirements Eliciation and Validation with Real World Scenes,* IEEE Transaction on Software Engineering, Vol. 24, No.12, December, 1998

[79] R. Dömges and K. Pohl, *Adapting Traceability Environments to Project-Specific Needs,* Communication of the ACM, Vol. 41, No. 12, 1998