UPGRADE
The European Journal for the Informatics Professional
http://www.upgrade-cepis.org
Vol. V, No. 5, October 2004

UPGRADE is the European Journal for the Informatics Professional, published bimonthly at <http://www.upgrade-cepis.org/>

UPGRADE is the anchor point for **UP**ENET (**UP**GRADE European NETwork), the network of CEPIS member societies' publications, that currently includes the following ones:
- Mondo Digitale, digital journal from the Italian CEPIS society AICA
- Novática, journal from the Spanish CEPIS society ATI
- Pliroforiki, journal from the Cyprus CEPIS society CCS
- Pro Dialog, journal from the Polish CEPIS society PTI-PIPS

Publisher
UPGRADE is published on behalf of CEPIS (Council of European Professional Informatics Societies, <http://www.cepis.org/>) by Novática <http://www.ati.es/novatica/>, journal of the Spanish CEPIS society ATI (Asociación de Técnicos de Informática <http://www.ati.es/>).
UPGRADE is also published in Spanish (full issue printed, some articles online) by Novática, and in Italian (abstracts and some articles online) by the Italian CEPIS society ALSI <http://www.alsi.it> and the Italian IT portal Tecnoteca <http://www.tecnoteca.it/>.
UPGRADE was created in October 2000 by CEPIS and was first published by Novática and INFORMATIK/INFORMATIQUE, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <http://www.svifsi.ch/>).

ISSN 1684-5285

Next issue (December 2004):
"Cryptography"
(The full schedule of **UP**GRADE is available at our website)

* This monograph will be also published in Spanish (full issue printed; summary, abstracts and some articles online) by **Novática**, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*) at <http://www.ati.es/novatica/>, and in Italian (online edition only, containing summary abstracts and some articles) by the Italian CEPIS society ALSI (*Associazione nazionale Laureati in Scienze dell'informazione e Informatica*) and the Italian IT portal Tecnoteca at <http://www.tecnoteca.it>.

Supporting the Software Process in A Process-centred Software Engineering Environment

Hans-Ulrich Kobialka

In software projects there exist tools, working schemes, and collaboration. Turning such an environment into a Process-centred Software Engineering Environment (PSEE), involves: 1) Augmenting the process with an additional flow of information and a dedicated user interface. 2) Supporting wanted process steps. 3) Disabling unwanted process steps. The reason for this is that it is not practical to claim that all activities in software projects have to be completely defined and supported by a PSEE. Instead, process support has to be introduced incrementally. This paper illustrates how this can be achieved.

Keywords: ALADYN, Process Modelling Language (PML), Process-centred Software Engineering Environment (PSEE), Software Process Enactment, Software Process Support.

1 Introduction

A software process encompasses a significant number of steps performed by many people. Several years ago it was stated that software processes should be described in a formal way by using Process Modelling Languages (PMLs), and that models written in a PML can be enacted in Process Centred Software Engineering Environments (PSEEs) [13].

Current PMLs/PSEEs cause difficulties when used during process enactment. This is because many PMLs/PSEEs have adopted approaches from specification and programming languages which were not designed to adapt to change during execution. Process programs often turn out to be quite complex which users have found difficult to both understand and apply changes to correctly.

Section 2 introduces the requirements which have to be fulfilled for effective process enactment. The concepts of current PMLs/PSEEs are discussed in Section 3. We then propose a number of ingredients which enable successful process support (Section 4). While this is done on a rather general level, in Section 5 we give a more technical view on Event-Condition-Action (ECA) rules and impact control.

2 Requirements of Process Support

Process support raises several challenges which have to be met by a PML, and its runtime environment the PSEE. Here we make some statements and identify the corresponding requirements.

a) Process support will never cover the entire process.

We suspect that support for a non-trivial process will never be complete in the sense that each possible sequence of actions is considered. This may be due to incomplete knowledge about the process, but also because process designers do not want to specify too much detail if there is no equivalent pay back in productivity. Therefore, only parts of the process which need support are specified in the PML – unsupported process steps are enabled but not explicitly written down on the level of process implementation (although they may be contained in handbooks or high-level process specifications).

Requirement 1) The PSEE should empower the user to work with incomplete process support.

It should be possible to perform working steps currently not specified or supported. In the extreme case, users should be able to work without writing any process program in advance. They should be able to build up working environments manually and to pass around results. This requires a number of *services* to be provided by the PSEE, e.g. communication services.

b) Process change during enactment is the normal case, not the exception.

Process support is likely to change during a process. It is usually impossible to define the entire software process support from the very beginning. In most processes only the principal aspects (e.g. some data & control flows) are known, but their volume or frequency cannot be determined in advance.

Software processes "run" for a long time. During this time things may change which again might force the process to be adapted or even changed completely.

Hans-Ulrich Kobialka got his MSc in Computer Science in 1985 at TH (*Technische Universität Darmstadt*), Germany, and his PhD in 1998 at TUB (*Technischen Universität Berlin*) Cottbus, Germany. Since 1985 he worked as a scientist at GMD and since 2001 at Fraunhofer AIS. His research interests include robot control architectures, real time computer vision, and software process support. <hans-ulrich.kobialka@ais.fraunhofer.de>

Requirement 2) The change of a process program has to be a simple and save operation.

Installing a change should be possible at runtime by the push of a button without the risk of any hazardous effects. There should be a means of defining and controlling permissions for process change.

In case a change has not achieved the expected benefits, it should be possible to undo that change easily.

c) Process programs cannot be assumed to be totally correct.

Process programs are difficult to test, because they operate on large databases, interact with many users, and are written by different authors. The impact of changes to process programs are difficult to predict. Thus, when a process program is recognized to be incorrect, the state of enactment (i.e. the states of all current processes) has to be updated as well:

Requirement 3) The user should be able to complement or compensate for the effects of process programs.

The interactive user should be able to perform every action that can possibly be performed by a process program. The service interface of the PSEE has to cater for basic constraints of consistency.

d) The probability of misunderstandings and errors is a key factor.

Although errors should be tolerable, they should be avoided as much as possible. Manual repair actions are time-intensive and can create further errors, especially when carried out by different authors. A certain frequency of errors can reveal a PSEE to be totally unusable in an industrial context.

Requirement 4) The effect of process programs should be easily understood by users.

The *impact* of a statement should be obvious to the reader. In particular, the side effects on other statements should be limited as much as possible. The *size* of the process support to be written should permit parts to be located easily and understanding of the support to be total. The PML should support *abstractions* of the software process domain. Appropriate abstractions may also reduce the size and complexity of process programs.

e) Process support will be more effective if it can be customized for each task.

The characteristics of tasks, and the skills of teams, can differ substantially. Much productivity (and acceptance of process technology) can be gained if a process can be adapted in such a way that these characteristics and skills can be exploited. Although projects interact with each other, they require some *autonomy* in adapting their processes.

Requirement 5) Process support should be adaptable to specific (sub)projects.

The effects of a process support, which has been installed for a particular task, has to be limited to this part of the project.

3 Related Work

PMLs are often classified into net-based, object-oriented or rule (or event-trigger) based approaches. Most PSEEs offer a hybrid PML with multiple paradigms (or multiple PMLs respectively), see Table 1. Due to space limitations, in this paper we discuss the above approaches rather than each individual system.

3.1 Net-based PMLs

Net-based PMLs (Process Weaver [8], Slang/SPADE-1 [1], LEU [9], Endeavors [2], APEL (the PML of the Adele PSEE) [6], Little-JIL [4]) have several advantages: they can be used to describe a process in an intuitive way, they can be analysed and simulated, and in most cases the impact of their elements (states, transitions) can be easily understood.

Net-based PMLs permit a transition to execute some code which may access any database and communicate with concurrent transitions or remote services. In these cases, the models are no longer genuine Petri-nets, because the latter requires the firing of a transition to depend solely on its inputs. Thus, simulation and impact analysis can become difficult as databases and communication are used in a concrete net.

The main drawback of net-based PMLs is that users tend to specify only the main workflows. Unexpected situations are difficult to handle if a net state has to change during execution. Some PSEEs (e.g. SPADE-1) permit changes at runtime, but this puts the user in the position of editing a net, i.e. still having to plan steps before executing them. If tasks have to be created and connected dynamically, net-based models offer no appropriate execution model.

3.2 Object-oriented Approaches

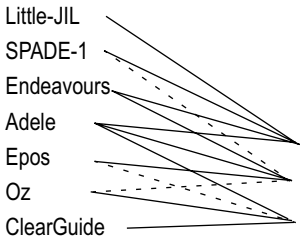
Object-oriented PMLs (EPOS [5], Adele) offer more flexibility, as they allow users to create and connect tasks (activities) dynamically (incremental evolution of task networks). Dynamic instantiation is also used in non object-oriented approaches, e.g. DYNAMITE, ClearGuide [12].

Many object-orientated approaches (e.g. EPOS, Adele, Endeavors, SPADE-1) make intensive use of the definition of object-oriented class hierarchies. We argue that this is not the right approach to implement and change process support during enactment because the units of change are schemas (type hierarchies) which tend to become quite large and contain complex dependencies. Therefore, schema updates can be difficult (especially if existing objects can be invalidated), and the transfer of improvements from one schema to a similar one is not trivial.

3.3 Rule-based PMLs

Rule-based and event-trigger based approaches are very powerful. However process descriptions tend to become complex, especially if rule chaining (e.g. in Marvel/Oz [3]), or other kinds of side effects between rules/triggers are used. APEL and ClearGuide do not encourage the user to use rule chaining. Usually side effects occur between rules.

Multiple processes. Different process instances can coexist and cooperate within most PSEEs. If rules of different process-



	Complexity & Impact Control	Task-specific customization	Flexible Datat and Control Flow	Change Effort
net-based PMLs	+	+[1] –	–	+ –[2]
object-oriented PMLs	–	–	+	–
rule-based PMLs	–	–	+	+

1. Task-specific support could be possible, if the PSEE allows to use different variants of a (sub)net within different tasks. Not all net-based PMLs allow this.
2. The effort can be reasonable if several instances of the net are currently existing. Each instance has to be stopped and investigated whether the placement or contents of the tokens conflict with the change of the net.

Table 1: The Strengths and Weaknesses of Different PML Paradigms.

es (possibly written by several authors) interfere or conflict, most PSEEs do not consider who has installed the conflicting rules. Only ClearGuide knows during execution of a rule, for which task this rule has been defined.

Control of Permissions. Context information (the current user, the task he is currently performing, and in the case of a rule the task/process for which it is defined) can be used for conflict resolution and access control. As PSEEs offer only limited context information, their ability to control permissions is also limited.

4 Ingredients for Process Support

Here we propose a number of ingredients for the design/architecture of PMLs and PSEEs in order to enable effective process support.

4.1 PML = Service Interface

As process steps should be carried out concurrently by process programs there has to be some service which is accessed by multiple processes and which coordinates their activities. If this service interface is also accessible via a command interface, or a graphical user interface (GUI), and if this interface offers an appropriate abstraction level, it fulfils requirement 1: the user can perform process steps even if a process programs is incomplete or does not exist. This also meets requirement 3: in case of an incorrect/incomplete process program, the user can manually repair the state of enactment.

The interface of process service defines the PML. The process server acts as an interpreter of this PML. The remainder of Section 4 considers the question "What are the right services for process support?".

4.2 Inter-related Process Objects

The PML services should provide basic concepts of processes, tasks, users and communication. It should be possible to dynamically create/connect/delete instances of these objects thereby managing networks of interrelated objects.

Process support should offer some notion of task (also called activity) and relationships between tasks. A special relationship is the task-subtask relationship: a task is divided in several subtasks; the manager of a task also heads (is the managers of) the subtasks. Users are assigned to tasks, thereby fulfilling special roles (e.g. manager). There should be some communication infrastructure e.g. one which permits a message to be sent to the manager of task X rather than to a specific user.

A task may produce documents as results. These documents may again be associated as inputs to other tasks. Documents are typically stored in some configuration management system (e.g. CVS, ClearCase, Adele).

4.3 Process Definition on the Instance Level

In a network of interrelated objects, additional process knowledge has to be added e.g. process procedures and constraints. Also the process can be parameterized using process variables, similar to environment variables e.g. a process variable may simply denote which editor should be invoked by default.

We propose that process variables, procedures and constraints should be contained in process programs which are associated with task objects.

Note that there is no need for type hierarchies, or other kinds of dependencies on the type level, which make change difficult. Processes are defined on the instance level: tasks are instantiated and specific process knowledge is added to these instances by assigning process programs to them.

4.4 Process Permissions

The creation of tasks, subtasks, and relationships to other objects is a privileged operation. The same holds for associating a process program to a task. Such operations need special permission.

We propose that such operations can only be performed by users who play a certain role ("manager") in a task. Of course, a manager can only perform privileged operations on his task and its subtasks. Because process procedures and constraints are associated to a task by its manager, they are performed with the same permissions (as if these actions were performed by the manager manually).

4.5 Process Automation

Active mechanisms, like the Model-View-Controller pattern, Event-Condition-Action (ECA) rules, triggers, or similar mechanisms e.g. in ClearGuide or APEL, have many advantages

- They are very powerful and flexible. E.g. it is very time-consuming to document all possible action sequences which may be caused by a few ECA rules.
- They decouple the event source from the action. There is no need to modify the event source if an ECA rule is added, modified or removed.
- They are quite robust with respect to side-effects from other components, *if they are written carefully*. ECA rules do not know when they will be executed. Other rules associated with the same event may be executed before, possibly modifying/deleting objects. An ECA rule has to check all its assumptions first before performing any action. In contrast to this, the 'sequential programming' paradigm is more sensitive to change. For example in the sequence "A; B; C;" (or "A→B→C"), statement C usually contains assumptions about the previous statements. So when changing statement A, all subsequent statements are possibly affected.

Constraints

Elaborated process services and active mechanisms offer a very powerful framework. But not every possible sequence of actions is desirable. Manual detection and repair of unwanted states is not feasible in an automated environment. So if a user issues a command which violates consistency constraints, this has to be aborted on the database level as if it was never executed.

The principle behind this is that there is some model defined which initially offers a great deal of freedom. As more knowledge about the process is obtained over time, that freedom can be restricted to exclude unwanted actions. According to Peter Wegner: *"Constraints are more powerful because they are applicable to non-compositional behaviours. Michaelangelo's sculptures realized by chipping marble slab could not have been realized by gluing together small bits of marble."*.

5 Process Automation with ALADYN

The above proposed ingredients for process support are described on a rather general level. In order to give a more concrete impression of how process support may look like, we describe the process automation and impact control in a concrete system. Here we refer to a PSEE named ADDD, which offers a PML called ALADYN [10].

Besides the organization of work, the task hierarchy is used to organize process descriptions, called ***policies***. Multiple aspects are grouped and treated together in a policy: *parameters* and *procedures* used for customization, and *triggers* used for process monitoring, notification, automation and consistency control.

A policy can be instantiated for several tasks (Figure 1). ***Instantiation of a policy*** means instantiation of all its parameters, procedures, and triggers for the denoted task i.e. the contents of the policy file is parsed, and objects are created and inserted into system tables. A policy, which is instantiated for a particular task, influences the execution of this task and its subtasks.

For a task at most one policy can be instantiated. The instantiation of a policy for this task implies that the parameters, procedures, and triggers of a former instantiated policy are removed from the system.

5.1 Triggers and Constraints

Instantiated triggers are Event-Condition-Action (ECA) rules as used in active database systems. They are used to implement reactive behaviour.

In the trigger syntax, the concerned tasks are specified by a task pattern.

Task Patterns

A task has a pathname which is generated from its name and the names of its parent tasks in the task hierarchy. Task pathnames are similar to those of files in a file system, e.g. /projectA/Test/test_release.2.5. Task patterns may contain wildcards e.g. as known from the Unix shell. For example, the pattern *test* may be used to match all test tasks in the system.

However, general pattern matching is not selective enough. Usually a trigger should be applied to tasks located in the context of the task for which the policy is instantiated.

Therefore, a pattern can contain one of the keywords SELF, PARENT, or PROJECT at the beginning. These keywords refer to the task for which the policy is instantiated (SELF), its parent task (PARENT), or the top-level task containing it (PROJECT). Each time an event is matched against the trigger, the keyword is replaced by the pathname of the denoted task. For example, if a policy is instantiated for the task /projectA/ Test and one of the policy's triggers has the task pattern SELF*, the pattern is expanded to /projectA/Test* (i.e. the pattern matches /projectA/Test and all its subtasks). Similarly, a trigger containing PARENT* looks at the task /projectA and all its subtasks.

"Happened on" versus "Caused by"

During command execution a task can be in two positions. First, it can be the active task on behalf of which the user issues a command and implicitly *causes* events. Second, a task can be a passive object which is modified by a command and on which events are *happening*.

ALADYN distinguishes between (1) triggers which wait for events happening on a particular task (i.e. the task is modified), and (2) triggers observing events caused by a particular task (i.e. the current execution context refers to this task).

trigger_on_task EV_NEW_RESULT <*task pattern*> route_result_to_review

This trigger defines that the action route_result_to_review is executed each time when the event EV_NEW_RESULT *happens on* a task matching the task pattern, i.e. when a result of this task is published. This trigger fires regardless of which task has caused the event. Such types of trigger are denoted by the suffix _on_task.

trigger EV_NEW_CR <*task pattern*> handleChangeRequest

This trigger says that each time a change request has been created (i.e. *is caused by*) the tasks matching the task pattern,

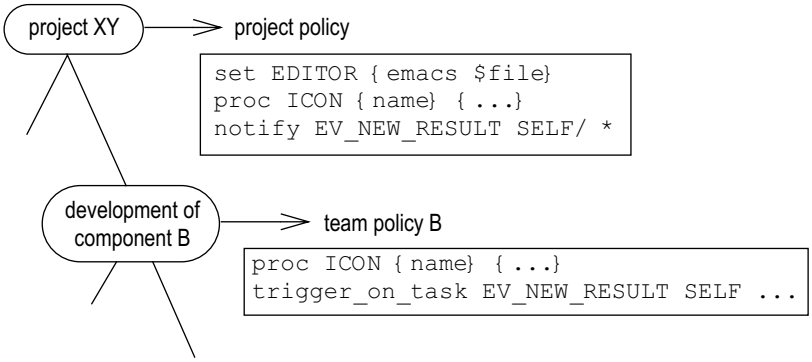


Figure 1: Example of Policies Defined on Different Levels in The Task Hierarchy.

an action (handleChangeRequest) should be executed automatically.

Optionally, the execution of triggers can be controlled by conditions, i.e. the action is only executed if the condition is fulfilled. Conditions and actions of triggers are procedures. When a trigger is defined in a policy file, its condition and action procedures should be written into the same file.

Constraints

In some cases, process support has to restrict certain actions in order to preserve consistency constraints. Usually it is very difficult to undo violations after they have occurred and other users have seen them or triggers have reacted on them. In ALADYN, constraint triggers can be used to abort modifications on the database level (including all triggered procedures). This resets the system to the state before the offending command was issued (i.e. as if it had never been executed).

The constraint and constraint_on_task trigger declarations are similar to trigger and trigger_on_task, except that the triggered action is implicitly "abort". For instance, the following trigger prevents the tasks matching the task pattern from assigning any policy file to a task:

constraint EV_NEW_POLICY_FILE <*task pattern*>

5.2 Controlling Impact of Process Programs

We illustrate how the impact of triggers, parameters, and procedures is limited to the task for which they are instantiated. This impact can be further restricted by permissions and constraints.

5.2.1 Impact of Process Parameters and Procedures

Process parameters and procedures defined in the policy of a task are inherited by its subtasks. For instance in Figure 1 the ***project policy*** is instantiated for task "projectXY" and defines that the emacs editor should be used. This parameter is inherited recursively by all subtasks of projectXY, i.e. everyone working in this project will use emacs.

On the other hand, parameters and procedures can be redefined in the policies of subtasks. This means that within the execution context of the subtask, the redefined value/implementation is retrieved for a parameter/procedure, while commands executed on behalf of the parent task still retrieve the value defined in its policy. In Figure 1 the procedure ICON is redefined by ***team policy B***. This leads to (some) items being displayed with different icons for people working on B than are displayed for people working on the rest of the project.

5.2.2 Permissions and Constraints

If a process procedure is invoked, it is performed within the execution context of the command, i.e. the procedure is executed with the same permissions as the invoking command.

A user can be assigned to a task either as a manager or a developer. A manager of a task has certain permissions to configure the task and its subtasks (creating subtasks, instantiating policies, assigning people, passing input to a task, etc.), but the manager is *not* allowed to do this for tasks located elsewhere in the task hierarchy. Thus the permissions of managers are limited.

Developers have no management permissions and work with the configuration management service (e.g. checkin/checkout of versions and configurations).

The permissions of managers and developers can further be restricted in a task-specific way by constraints ***if*** these constraints are defined by one of their managers, as explained below.

5.2.3 Impact of Triggers

In contrast to parameters and procedures, a trigger which is instantiated for a task cannot be inherited, redefined or disabled by a subtask. For example, in Figure 1 the ***project policy*** defines that the team members of project XY should be notified whenever one of its subtasks produces any results. This trigger is not implicitly instantiated for subtasks, i.e. task "development of component B" is *not* notified if one of its subtasks produces a result.

The impact of a trigger can be determined first of all by its task pattern. This defines the tasks which are viewed or controlled. Other parts (the event, the condition and the action procedures) also restrict the impact, but the task pattern is usually the most selective part of a trigger i.e. it filters most of the events occurring in the system.

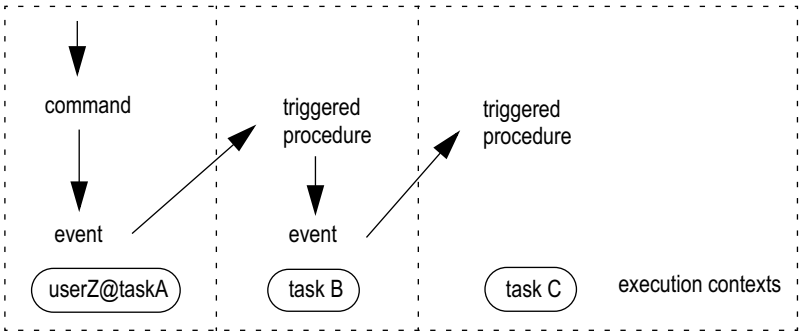


Figure 2: The Execution Context Is Switched for Each Triggered Procedure (the arrows denote the flow of control.)

Permissions of triggered procedures:

A command can raise several events which may cause triggers to be fired i.e. procedures are executed. For each triggered procedure the context is switched i.e. the procedure is executed in the context of the trigger's task (Figure 2).

A policy can only be instantiated for a task by a manager. Therefore the triggers are authorized by this manager and accordingly have the same permissions. For example, a triggered procedure can only abort a command of userZ@taskA if the constraint is declared in a policy of taskA or one of its parent tasks.

5.3 Abstractions appropriate for the Process Domain

A clear understanding of the impact of each statement is an important factor in understanding the whole program. In addition, the size and the complexity of the program are also significant factors.

Beside general ECA rules, ALADYN offers the definition of *specialized triggers*. For example, the following trigger defines that a notification should be sent to the managers of the policy's task each time one of its subtasks produces a result.

```
notify EV_NEW_RESULT SELF/* MANAGER
```

Such events can also be logged in a file:

```
log EV_NEW_RESULT SELF/* $TEMP/taskIO.log
```

Specialized triggers offer abstractions which are appropriate for the process domain. While ECA rules stem from active databases, concepts like notification, logging, and configuration management belong to the process domain. Such abstractions free the domain expert from writing domain-specific solutions in terms (and many lines) of a general-purpose language.

We have implemented support for the ISPW9 scenario [14] in ALADYN [10]. The solution contains 9 triggers which invoke procedures, and 19 "trivial" triggers which do not. Only one trigger causes another "non-trivial" trigger to fire, so programming required some care in this case. This solution has moderate size and complexity, although we have aimed for a comfortable level and not minimal support.

6 Conclusions

Effective process support depends on several ingredients such as high level services, user interfaces to these services, process definition on the instance level, active support for automation and consistency control, clearly defined impact, and control of permissions.

In this paper we discuss a number of requirements, and some problems existing PMLs have in meeting these requirements. We then propose a set of ingredients which we regard as helpful, or even mandatory, to remedy the problems. Because this is done on a general level we then illustrate some aspects by describing process automation using the ALADYN PML.

The ALADYN PML contains no mechanisms to enforce control flow explicitly as can be done with net-based PMLs. Instead ALADYN supports the flow of results and automatic notification, and lets the user decide what to do next. Control flow enforcement requires the programming of ALADYN constraints. An interesting approach is to define control & data flow in a net-based language and then generate PML code for the PSEE to be used. In [7], process descriptions defined in UML were used to generate code for the OPSS PSEE. Such an approach can be used to generate support for the specified process without prohibiting non-specified processes. This combines the benefits of precise specifications with the flexibility of an interpreted PML.

Because of space restrictions, this paper does not discuss how process support can be improved by process-specific user interfaces. For this important point we refer to [11].

References

[1] Sergio Bandinelli, Elisabetta Di Nitto, and Alfonso Fuggetta. "Supporting cooperation in the SPADE-1 environment." IEEE Transactions on Software Engineering, 22(12):841–865, December 1996.

[2] Gregory Bolcer and Richard Taylor. "Endeavors: a process system integration infrastructure." In Proceedings of the 4th International Conference on the Software Process, pages 76 – 89. IEEE Computer Society Press, 1996.

[3] Israel Ben-Shaul and Gail Kaiser. A Paradigm for Decentralized Process Modeling. Kluwer, 1995.

[4] Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, Stanley M. Sutton, Jr., and Alexander Wise, "Little-JIL/Juliette: A Process Definition Language and Interpreter", In

Proceedings of the 22nd International Conference on Software Engineering, pp. 754–757, June 2000.

[5] R. Conradi, M. Hagaseth, J. O. Larsen, M. N. Nguyen, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. "EPOS: Object-oriented cooperative process modeling." In B. Nuseibeh, A. Finkelstein, and J. Kramer, editors, Software Process Modelling and Technology, pages 33–70. John Wiley and Sons, 1994.

[6] Samir Dami, Jacky Estublier, and Mahfoud Amiour. "APEL: A graphical yet executable formalism for process modeling." Automated Software Engineering, 5(1):61–96, January 1998.

[7] Elisabetta Di Nitto, Luigi Lavazza, Marco Schiavoni, Emma Tracanella, and Michele Trombetta: "Deriving executable process descriptions from UML", In Proceedings of the 24th International Conference on Software Engineering, pages 155–165. ACM Press, May 2002.

[8] Christer Fernström. "Process WEAVER: Adding process support to UNIX." In Proceedings of the Second International Conference on the Software Process, pages 12–26. IEEE Computer Society Press, February 1993.

[9] Volker Gruhn and Stefan Wolf. "Software process improvement by business process orientation." Software Process–Improvement and Practice, Pilot Issue:49–56, August 1995.

[10] Hans-Ulrich Kobialka. "Implementing support for software processes in a process-centered software engineering environment." Ph.D. Thesis. GMD Research Series 15/1998. <http://www.ais.fraunhofer.de/~kobi/img/SupportForSoftwareProcesses.pdf>.

[11] Hans-Ulrich Kobialka and Claus Lewerentz. "User interfaces supporting the software process." In Proceedings of the 6th European Workshop on Software Process Technology, Lecture Notes in Computer Science, September 1998. <http://www.ais.fraunhofer.de/~kobi/img/ewspt6.3.pdf>.

[12] David B. Leblang. "Managing the software development process with ClearGuide." In Software configuration management: ICSE 97 SCM-7 Workshop, pages 66–80. Lecture Notes in Computer Science 1235, Springer, May 1997.

[13] Leon J. Osterweil. "Software processes are software too, revisited." In Proceedings of the 19th International Conference on Software Engineering, pages 540–548. ACM Press, May 1997.

[14] Maria H. Penedo. "Life-cycle (sub) process scenario." In C. Ghezzi, editor, Proceedings of the 9th International Software Process Workshop, pages 141–143. IEEE Computer Society Press, October 1994.

