

Deriving executable process descriptions from UML

E. Di Nitto, L. Lavazza, M. Schiavoni, E. Tracanella, M. Trombetta

CEFRIEL – Politecnico di Milano
Via Fucini, 2
20133 Milano Italy
+39 2 239541
{dinitto, lavazza, tracanel}@cefriel.it

ABSTRACT

In the recent past, a relevant effort has been devoted to the definition of process modeling languages (PMLs). The resulting languages and environments –although technically successful– did not receive much attention from industry. On the contrary, researchers and practitioners have recently started experimenting with the usage of UML as a PML. Being so popular and widely used, UML has an important competitive advantage compared to any specialized PML. However, it has also a main limitation. While most PMLs are executable by some process engine, UML was conceived as a non-executable, semi-formal language. The work described here aims at assessing the possibility of employing a subset of UML as an executable PML. The article proposes a formalization of the semantics of the UML subset and presents the translation of UML process models into code, which can be enacted in the OPSS process-centered environment. The paper also presents a case study to validate the approach. We expect that process modeling by means of UML is easier and available to a larger community of software process managers. Moreover, process enactment makes the process more efficient, reliable, predictable and controllable, as widely shown by previous research.

Categories and Subject Descriptors

D.2.9 [Management]: Software process models

General Terms

Management, Documentation, Design, Languages.

Keywords

Process modeling, process support systems, UML.

1. INTRODUCTION

Research on software process and workflow management systems has produced several languages and systems particularly suited for process (or workflow) description and execution. Process Modeling Languages (PMLs) are similar in nature to programming languages (in [16] it has been argued that “software processes are software too”) and are usually executed by process engines, i.e., software systems which, given a process description, are able to partly automate and support the execution of the corresponding process. OPSS [6] is one of such process support systems (see Section 2).

In this context, several process modeling paradigms have been

proposed and used, including Petri nets, grammars, rule-based systems, etc. These languages and the related supporting systems (usually capable of enacting the process) did not receive much attention from industry. The main causes were the difficulty of describing process models using these languages and the lack of support for distributed processes. Recent proposals (like OPSS and several others) have solved part of the problems of early process-centered environments, like the support of distributed development processes, the integration with tools and the coordination of process agents. Nevertheless, they are hardly used in industry.

Workflow management systems (which are conceptually very similar to process modeling and enactment systems) gained a bigger popularity by relying on simpler PMLs, but their diffusion is limited to application domains where processes are very simple and quite repetitive.

Recently, UML has been also used as a PML [8], [11], [12], [13]. In fact, UML has some attractive features as a PML: it is popular, standard, graphical, equipped with several diagrams which support several views of the systems, extensible, supported by tools, provides a standard textual output (XMI), ... For instance, its authors used it to represent the Rational Unified Process (RUP) [11], which is recommended for UML-based developments. However, UML has been conceived as a non-executable, semi-formal language. In other words, UML process models are suitable descriptions for humans, while they are neither sufficiently precise nor detailed to be interpreted by machines.

The objective of this paper is to explore the possibility of using UML as a language to describe processes not just for usage by humans, but also for process enactment. As the target process environment, we have adopted OPSS (because we know it well) though in principle we could have employed other process-centered environments. The goal is to allow process modelers to write process descriptions by means of UML, and then convert these models into enactable OPSS models. By doing so we provide UML with a precise operational semantics and we make it enactable, while preserving its expressiveness as a high-level, human-oriented PML. The consequence is that process modeling and enactment becomes available to all the numerous process managers who master UML. As a side effect, we also solve the problem that OPSS, like several other PMLs, needs low-level programming for specifying all the details of the process. Finally, the automatic translation of UML into the OPSS Java classes makes the approach quite efficient.

The paper is organized as follows. Section 2 gives a brief description of OPSS. Section 3 presents how UML is currently used as a PML in some approaches related to the one we present. Section 4 presents the UML constructs we use to model processes and the semantics of such constructs. The translation of UML models into OPSS classes is described in Section 5. Section 6 presents the case study we have carried out to validate our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference '00, Month 1-2, 2000, City, State.

Copyright 2000 ACM 1-58113-000-0/00/0000...\$5.00.

approach. An evaluation of the work done is reported in Section 7. Finally, Section 8 draws the conclusions.

2. A BRIEF INTRODUCTION TO OPSS

OPSS is a workflow management system we have developed in the past years [6], [7]. It has been conceived to support the design and operation of sophisticated process-based services like electronic commerce, customer care, remote education, and software development. Differently from traditional process-based activities that are limited to the boundaries of a single company, where the number and the identity of actors are quite stable, the processes we consider involve a high, extremely variable number of distributed concurrent users. Based on this consideration, we tried to develop OPSS in such a way that the execution of the various process activities is distributed over a number of agents, and the components of the system are loosely coupled with each other.

In OPSS, the *activities* that constitute a process can be executed by human beings or by some computerized equipment (often by a combination of the two). The executors of process activities are collectively called *agents*. Each agent receives an *activity description* (i.e., a process model fragment) and executes it. An activity description may be specified in any language that can be understood by the agent in charge of executing it. For instance, a human agent could execute activities expressed in some natural language. Software agents can either be generic executors of activity descriptions or specialized tools that encapsulate the semantics of a specific activity.

Figure 1 shows the high level architecture of OPSS. The figure shows three agents, one human agent and two software agents. The human agent interacts with the system through an agenda. The agenda informs the agent of all activities he/she is assigned to, and it allows the agent to inform the system of the status of each activity of his/her competence.

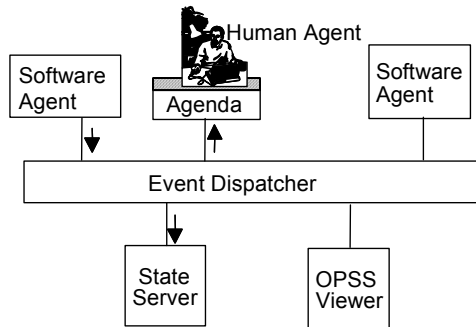


Figure 1: The ORCHESTRA Process Support System.

The figure also shows the OPSS Viewer and the State Server. The first one is a monitor that provides information about the state of the process. The State Server is a persistent repository that reifies the process state by mirroring the state of all its *process entities*. Such entities have been classified by the Workflow Coalition in [18]. Elaborating on such classification we came out with the following entities: *activities* that use and modify *artifacts*, and *software resources* that can be limited in number (e.g., a tool with a limited number of licenses) or unlimited. Activities can be composed of other sub-activities and can be related through precedence constraints. Finally, they can be automated (in this case we call them *software activities*) or not. *Agents* executing activities can be human beings, software components, or groups

of human beings. These entities are represented in the State Server as a set of objects, called *process entity representatives*, each containing a detailed description of a specific process entity. Figure 2 shows process entity representatives together with the relationships holding among them.

The State Server and the process entity representatives participate in the process actively: they monitor process execution and react to state changes according to some process rules that are encapsulated in associated state machines. Thanks to this characteristic, the State Server acts as a coordinator between agents that execute different interdependent parts of the process and monitors for any violation of the process constraints.

The State Server and all other components communicate mostly by exchanging events. The event dispatcher shown in Figure 1 is part of the JEDI middleware [6]. It manages the publication of events requested by OPSS components by dispatching them to all subscribers.

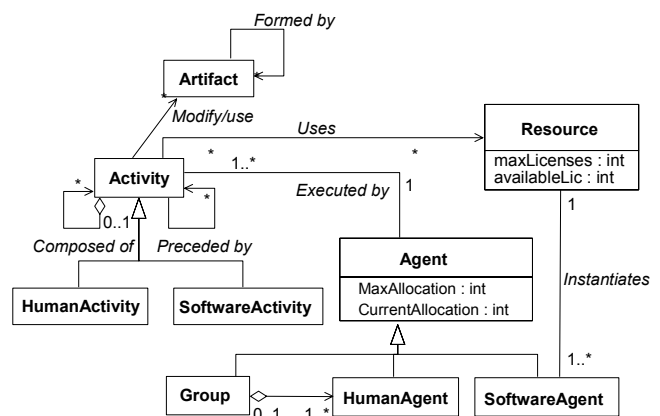


Figure 2: Process entity representatives and the State Server structure.

The State Server subscribes to events such as user logins or the creation of new activities, artifacts, or resources. When one of these events occurs (e.g., a new activity needs to be started), it creates the corresponding process entity representative. As mentioned before, process entity representatives react to events according to rules defined in a finite state machine that we call *life cycle*. A transition in the life cycle is defined by a triple: triggering event, condition, and action (they are similar to ECA rules in active databases). When an object receives an event E in a state S , all the transitions having S as initial state and E as triggering event are evaluated for firing. One of the transitions whose condition evaluates to true is non-deterministically fired, thus causing the execution of the action part and moving the instance to the target state. The execution of the action part can produce new events that may, in turn, affect the behavior of agents and the state of other objects in the State Server.

As an example, Figure 3 shows the life cycles associated to entity representatives *Activity* on the left and *Agent* on the right. Upon creation, the state of an instance of class *Activity*, AI , is set to *Inactive*. In this state AI is characterized solely by a unique identifier and by an activity description. AI can enter state *Assigned* when it receives event *AgentAllocated* (AI , $agentID$), i.e., an agent has been selected to execute the activity. The transition to state *Assigned* can only be executed if the instance of class *Agent* representing agent $agentID$ is in

state Available. In the current prototype, agendas subscribe to event AgentAllocated to provide human agents with information about their assignments. In the state Assigned, when AI receives event Activate(AI, activityID), it checks if the preceding activities have been terminated. If this is the case, it moves to state Active, and produces event ActivityStarted(AI, AD-URL). This event must be subscribed by the agent assigned to the activity or, if he/she is a human agent, by his/her Agenda. Parameter AD-URL contains the location of the activity description to be executed. If for any reason activity AI cannot be started when event Activate is received (e.g., it has to wait the termination of some other activity), it produces an event to warn the requesting agent.

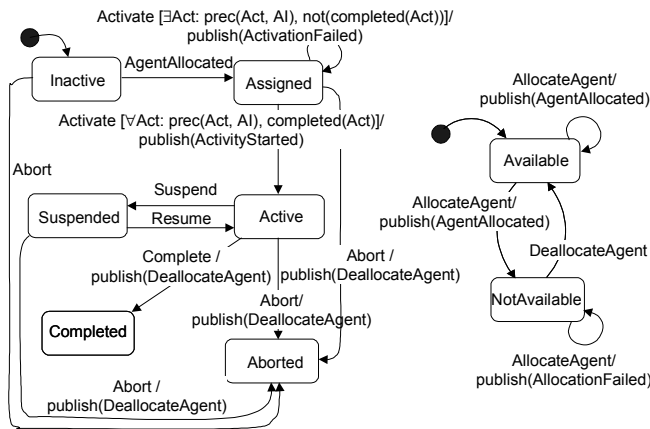


Figure 3: The Activity and the Agent life cycles.

Process entity representatives and the corresponding life cycles are implemented in OPSS as Java classes. Therefore, the behavior associated to each state transition is defined in the corresponding code.

Any OPSS component can query the state of the running process (i.e., of the process entity representatives) thanks to a set of RMI services provided by the State Server.

In the first release of the OPSS environment, process modeling was supported at a very low level of abstraction. The environment contained the Java classes of the core process entity representatives and the process modeler was expected to specialize them (i.e., derive new Java classes from the core ones) depending on the specific features of the process being modeled. When needed, the process modeler had also to write the code for the activity descriptions executed by software agents.

3. UML AS A PML

The main purpose of the Unified Modeling Language (UML) [15] is to describe software systems at different levels of abstraction. It provides a number of different constructs and diagrams that enable the system modeler to define different views of a system and of the corresponding domain. Since most information systems are used to support business processes within organizations, a part of UML has naturally evolved to become a simple PML, thus allowing system analysts to formalize the processes to be automated or supported by using the same linguistic means they normally use to describe the other parts of the application domain. In UML processes are described through *activity graphs*, that, in turn, are syntactically expressed in *activity diagrams*. For the sake

of simplicity in the following we do not make any distinction between these two terms.

Based on the UML specification, some approaches have been recently proposed in the literature, which provide guidelines to use the diagrams/constructs already available in UML, as well as to extend them for process description. One of these approaches – presented by Eriksson and Penker in [8]– defines a number of stereotypes and constraints that specialize the way the standard constructs are used. Most notably, they introduce in activity diagrams the concept and the stereotype of *process*, i.e., an activity that can be further decomposed in sub-activities and can span over multiple swim lanes to represent the fact that multiple roles can cooperate to its execution. In addition to activity diagrams, authors suggest to use stereotyped classes and class diagrams to describe resources, roles, business goals, events, etc. Finally, they suggest structuring the process description in four views at different levels of abstraction, the *business vision view* in which processes are formalized mainly through activity diagrams, the *business structure view* that describes the organization of resources and information being used and/or produced in the process, the *business behavior view* that illustrates the individual behavior of resources and processes and the interaction between them. Authors also provide guidelines and patterns to support the modeling activity. Both the language extensions and the supporting methods have been designed with the purpose of eliciting and describing processes that are to be understood and interpreted exclusively by human beings. To this end, the focus here is to provide an intuitive and rich linguistic support, while formality and non-ambiguity of each construct is not so important since it is left to the interpretation abilities of humans. Under this respect our goal is different, since we want to be able to enact a described process. This results in the need for associating a precise operational semantics to all linguistic constructs. Also, we want to avoid the definition of new stereotypes or other extensions for UML in order to simplify the work of the process modeler, and to make our approach readily usable by anybody who knows standard UML.

Another proposal that exploits UML as PML is presented in [12]. Here the goal is to obtain an enactable description. To this end, authors select class and state diagrams as main constructs to describe processes. In these diagrams, activities (tasks) are represented as “task packages” and “realization packages”. A task package encapsulates the interface of a task, i.e., its behaviors as seen by an external observer. A realization package defines how the task is realized in terms of other lower level tasks. In the corresponding class diagrams, the input and output of each task is explicitly shown. Flow of control and data between tasks are represented through properly stereotyped relationships. Collaboration diagrams are used to describe evolution scenarios for the process. The internal behavior of tasks is described by a predefined and un-modifiable state diagram that makes the task evolve through the states *InDefinition*, *Waiting*, *Planning*, *Active*, *Suspended*, *Done*, and *Failed*. Compared to the previous approach, this one is clearly more focused on adapting UML to the capabilities and semantics of the virtual machine that will be used to enact the process. Therefore, the process is described at a low level of abstraction. Some aspects –that are described in the basic UML approach and in the Eriksson and Penker extension– in this case are not expressed in the process modeling description,

probably because they can be inferred from the semantics of the underlying virtual machine. For instance, it is not apparent how the roles that participate in the process are described and how they are associated to the various activities to be executed, how the relationships between the data used and/or produced in the process are described, how possible parallelisms between activities, synchronizations and decision points are expressed. This, together with a massive usage of stereotype icons that are new to the reader, makes the resulting process description difficult to be understood by a human being, and quite different from our UML-based approach.

4. UML AND OPSS

UML appears to be a good candidate for a PML as it is graphical, intuitive, and easy to be understood. In addition, it is object-oriented and models behavior by means of state machines: this makes relatively easy to map UML constructs onto the concepts supported by most PMLs. In the case of OPSS, which employs an object-oriented language (Java) and models the evolution of the process by means of a finite state machine, the mapping is –at least conceptually– straightforward.

In fact, by properly restricting the usage of UML and providing a clear operational semantics for each process construct in UML, we obtain UML models which can be translated into sets of OPSS classes, thus enabling the enactment of the process being defined.

By taking this approach we do not intend to sacrifice the expressiveness of UML just to achieve simplicity of translation. In other words, we try to pose as little constraints and limits as possible to the use of standard UML.

By staying close to standard UML we also achieve an important benefit. In software development UML models are built to concentrate on design issues and then translated into code and enriched with implementation details. Likewise, the user of our approach builds high-level UML process models (concentrating on relevant process issues), and then translates them into code. In this way the need to write Java code in order to enact OPSS process models is minimized (and often completely eliminated), thus making process modeling easier and faster for managers and all the people who typically need process models, but do not have enough time or programming skills to write detailed models.

In this section we intuitively present the way we use UML for process modeling. A formalization of the constructs, extensions and constraints adopted is given in Z [17] and it is not reported here because of space constraints.

The diagrams we adopt for process modeling are the activity, class, and state diagrams. In particular we use the activity diagram, as usual, to describe the flow of activities in the process and the association between activities and agents executing them. A few refinements of the constructs of activity diagrams are presented later in this section.

We use class diagrams to associate concepts belonging to the level of the process description with concepts that are part of OPSS and that are reified in the State Server. In particular, we provide the process modeler with a number of predefined classes that describe in UML the process entity representatives (activities, agents, resources, artifacts, and their subclasses). Such classes have associated a state diagram that describes in UML the life cycle of each entity. A process modeler willing to use OPSS has to start defining his/her own activity types, agent types, etc. by

specializing the existing classes. Via specialization the modeler can modify the default values of the attributes of the base classes, or introduce new operations. For instance, Figure 8 shows a fragment of class diagram where a specialization of a `HumanActivity` is shown. Such specialization describes a testing activity performed within the context of the software development process presented in Section 6. Notice that a new attribute and a new operation are introduced.

At the level of the class diagram, the process modeler can also define “part of” relationships between activities and the corresponding subactivities, or between groups and the agents that compose them. Also, he/she can decide to specialize the state diagram inherited by the base class. Before focusing on state diagram specialization, let us provide some details on the structure of OPSS-compatible state diagrams. Each transition in a state diagram must be adorned with the following elements:

`InputEvent(arguments)[Guard]/[Action]/[OutputEvent(arguments)]`

`InputEvent` is the name of the event that will trigger the transition. During process execution, such event is then reified in a JEDI event. The specification of the event is mandatory: otherwise at run time it would be impossible to determine when the transition has to be actually triggered. The specification of the event may include a number of parameter names, which have to correspond to some attributes in the class definition. For instance, in the fragment of state diagram associated to activity `Test` shown in Figure 4 the input event called `SuspendActivity` has an argument, `instanceNumber`, which corresponds to an attribute of class `Test`. At runtime, given an instance of class `Test`, the event `Suspend` triggers the corresponding transition if and only if the argument `instanceNumber` associated to the event is equal to the value of the same attribute of that specific instance.

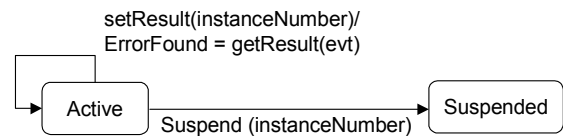


Figure 4. A state diagram fragment of activity `Test`.

All remaining elements belonging to the specification of a transition are optional. They are:

- A Guard: a boolean expression over the current state of the associated entity and/or the structure of the `InputEvent`. The Guard can contain calls to operations defined as part of the entity itself.
- An Action containing any Java statement, including a call to an operation defined as part of the entity. Such operation can have one or more parameters as usual. One of these –denoted by the name `evt`– can be the input event that has triggered the transition. Clearly, the signature of the operation, as it is described in the associated class description, has to be coherent with the call. The type associated to events in the signature and the body of the operation is `JEDIevent`. Figure 4 shows an example of action where a value is assigned to attribute `ErrorFound`. Such a value is computed by calling the `getResult` method.
- An `OutputEvent`. At runtime this is generated as a result of a state transition. In this case the argument list contains expressions whose values are transmitted with the event.

Each state diagram is required to contain an initial state, denoted according to the usual UML notation, and may contain one or more final states. At runtime, whenever an entity gets to its final state, it is eliminated from the state server.

Being a general-purpose language, UML does not define specific rules for state diagram specialization. Indeed, in the specification of UML version 1.3 three possible specialization approaches are described [15]. Among them we chose the method called *subtyping*: intuitively, given a state diagram A, the state diagram B can be considered as a specialization of A if:

- B defines new states and transitions, but it does not eliminate the states and transitions existing in A. For instance, in Figure 4 the state diagram of `HumanActivity` has been specialized with respect to Figure 3 by adding the transition from the state `Active` to the same state.
- B refines one or more states of A by means of proper state diagrams. For instance, Figure 5 shows the specialization of a fragment of the state diagram associated to `Agent`. Such a specialization is associated to `HumanAgent` and accounts for the fact that a human being, besides being available or not to take care of an activity, may also be temporary not logged into the system. Notice that when a state is expanded, the transitions associated to it can be specialized to identify the substates they are connected to: see for instance the transition triggered by the event `DeallocateAgent`. Since we assume that an agent can terminate an activity only if he/she is working, he/she has to be logged to issue the completion of the activity that, in turn, triggers this event. In Figure 5 `ra` stands for requested allocation or released allocation, `ca` for current allocation, `Ma` for maximum allocation. The last two ones are attributes of `Agent` (see Figure 10).

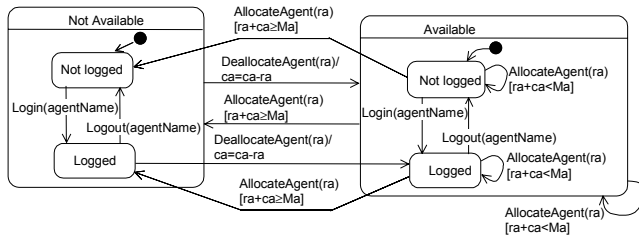


Figure 5. Specialization of the Agent state diagram.

The structure of the transition, i.e., the input and output events, the guard and the action cannot be modified in a specialization.

Figure 7 shows an example of activity diagram that describes the way the change management process illustrated in Section 6 works. In the current section we focus on the usage of UML more than on the example itself. All activities occurring in the diagram must correspond to some subclasses of `Activity`. Correspondence is checked through name matching. Swim lanes identify the agents executing the activities. In the class diagram, they have to correspond to specializations of class `Agent`. Activities can be executed in a sequence (see for instance `OrganizeCCB` that is followed by `ControlMRs`), that is, whenever an instance of the first activity is completed (its state diagram has reached the `Complete` state) an instance of the second activity is allowed to start. Notice that while in the normal UML semantics an atomic activity is executed synchronously with respect to its activation, in our case after activation the activity is

executed when an instance of agent is assigned to it and decides to take care of it.

The semantics of join and fork is enriched to describe the possibility of having multiple instances of the same activity type that are enabled for execution in parallel. This is achieved by properly annotating the transition connecting the fork to the activity type. Figure 7 shows two forks (F1 and F2) which create multiple instances of activities `AnalyzeAR` and `ImplementMR`, respectively. The number of created instances depends on the value of attributes `NumDesigner` and `NumImplementation`, owned respectively by the classes located before each fork. Consistently, in the context of a join it is possible to specify the number of instances that have to be completed before the activity following the join is activated. The keyword `ALL` indicates that all activity instances existing when the join condition is evaluated have to be completed.

In standard UML it is possible to indicate that an activity can have multiple instances by specifying the maximum number of instances in the multiplicity of the activity. It is not clear, however, when such multiple instances are activated and who takes care of this. In our interpretation we enforce multiple instances to be activated only as a result of a fork construct. By describing the number of instances to be activated at the level of the transition instead than at the level of the class –as in UML– this feature is made more evident to the reader.

UML does not pose any specific restriction to the conditions associated to the branches of decisions. Therefore, it is possible that more than one condition at a time is true, thus leading in a situation where multiple activities can be triggered. Also, it might happen that none of the conditions is true, and the execution of the process cannot proceed. In our approach we ensure that in the first case only one of the activities that can be triggered is non-deterministically enabled. As to the second case, we require that the process modeler always defines an “else” branch that is executed when none of the explicitly expressed conditions is true.

Decisions can be followed by merges. By means of decisions and merges it is possible to model cyclic operations as in Figure 6.

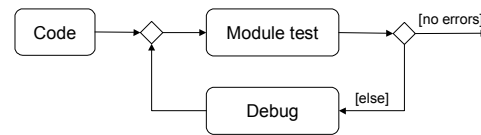


Figure 6. Definition of cyclic operations.

The UML specification suggests not to define input events for transitions in activity diagrams. In fact, transitions are usually supposed to be executed at the completion of the activity they originate from. This semantics, however, does not capture the cases, quite usual in real processes, when a new activity is spawned by another one that indeed does not terminate its execution. In order to address this requirement, we allow the process modeler to associate events to transitions. The semantics is that as soon as the activity that is the origin of the transition generates the specified event, an instance of the activity that is the destination of the transition is generated. In order to guarantee that the process can be enacted in the OPSS environment, the event triggering the transition has to be a JEDI event. It has to be explicitly described in the state diagram associated to the activity

as an event generated by some transition starting and ending in the Active state (or in any of its sub-states).

In UML, activities (i.e., instances of class Activity) can be refined in a sub-activity diagram. This is an interesting feature that enables process modularization and encapsulation. Unfortunately, we have realized that sub-activity diagrams are constrained to be contained in the same swim lane as the activity they refine. This means that it is not possible, or at least not immediate, to express that an agent delegates the execution of part of an activity to another agent. We could not understand if this is a limitation of the UML semantics or of the modeling tools we have used. In order to work around the problem, we have decided not to use the sub-activity diagram construct and to express the relationship between an activity and its sub-activities through proper associations at the level of the class diagram where activity types are defined. These relationships are not apparent within the activity diagram, thus any role in the process can own a sub-activity. The constraint that –consistently with OPSS semantics– is implicitly enforced in all activity containment relationships is that the main activity terminates only after that all the sub-activities are completed.

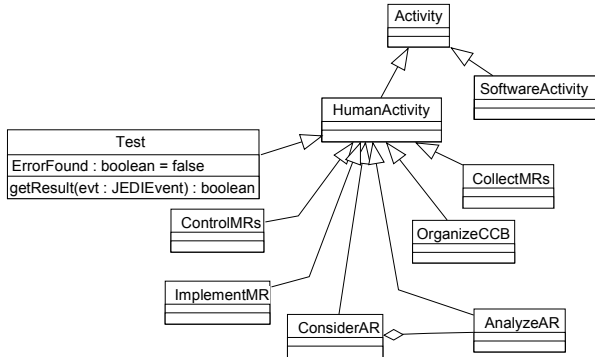


Figure 8. An example of activity hierarchy.

5. TRANSLATING UML INTO OPSS

The automatic translation of a UML process model into an OPSS model allows the process modelers to reason at a relatively high level of abstraction, while a translator produces models that are ready for enactment.

5.1 Translation Principles

The input to the translation is a UML model that respects the constraints and rules presented in Section 4. The output is the set of Java classes that compose the core of an OPSS model. The translation exploits the common features of the source and target languages, namely object-orientation and the description of dynamic behavior based on state machines.

The translation is executed successfully if the UML model conforms to the rules defined in Section 4. Violations of such rules are diagnosed by the translator. The translator also performs common semantic checks, for instance that an attribute appearing

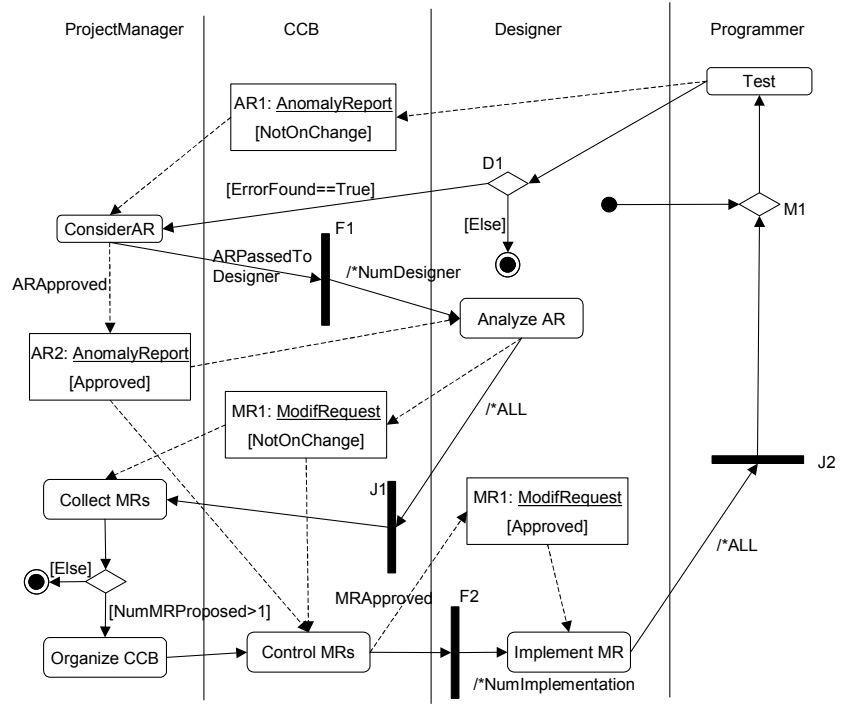


Figure 7: Main activity diagram describing the case study.

in a guard condition has been defined and is accessible. The translation proceeds according to the following simple criteria:

- Predefined classes (activity, artifact, agent, resource, etc., as shown in Figure 2) do not need to be translated, because the corresponding OPSS classes are already implemented.
- User-defined classes derived from predefined classes to describe specific elements of the process being modeled are translated into the corresponding Java classes, equipped with attributes, methods and associations as described in the given UML class diagrams. The body of new methods is defined according to the information provided by the corresponding state diagram and activity diagram (see the example of translation shown in Section 5.2).
- In particular, state diagrams are mapped into the life cycles associated to the OPSS classes. In such a mapping, nested states have to be properly flattened. For instance, super-state Available associated with a human agent (see Figure 5) is decomposed into states Available/Not logged and Available/Logged.
- Other relations represented in the activity diagrams (such as precedence between activities) are translated into Java code which manages such relations.
- The code of new operations introduced in user-defined classes can only be inferred from the state diagram of the class. If the modeler does not provide enough information in the state diagram, the implementation of the operation is left incomplete: the process modeler is supposed to complete this implementation after the translation has been performed.

Because of space constraints, here we do not give a detailed explanation of the translation criteria; instead we provide an example that illustrates how the translation is carried out.

5.2 An example of translation

Let us consider entity `Test`, taken from the case study reported in Section 6. The class and state diagrams of class `Test` are reported in Figure 8 and Figure 4 respectively. Fragments of the Java code generated for such entity are reported below.

```
public class Test extends HumanActivity {
    /* main attributes inherited by
       the super class
       public String CurrentParent = "";
       private static int InstanceID = 1;
       public synchronized static int
       getInstanceID(int Inc) {
           int temp = InstanceID;
           InstanceID = InstanceID + Inc;
           return temp;
       }
       protected int InstanceNumber;
    */
    public static String agentType =
        example.Programmer;
    public boolean ErrorFound = false;
    public string getResult(JEDIEvent evt) {
        return evt.getAttributeValue("RESULT");
    }
}
```

For the sake of completeness we have added to the code a comment that shows some of the attributes inherited by the super-class. `CurrentParent` indicates the activity that has originated `Test`. The value of such attribute is known only at run-time and is initialized by the constructor method of the activity. Attribute `InstanceID` and the corresponding method `getInstanceID` are used to keep track of the number of instances of the class that are created during process execution and to ensure that to each of them is assigned a unique ID. `InstanceNumber` contains the ID of a specific instance.

The value of the attribute `agentType` (in this case `Programmer`) is inferred by the swimlane in which activity `Test` is located (see Figure 7). The attribute `ErrorFound` and the operation `getResult` have been discussed in Section 4.

The code of class `Test` continues with the definition of the constructor. It is mainly concerned with the construction of an array that encapsulates the information about the state machine associated with objects of class `Test` (the starting and ending states and the type of event that can trigger the execution of the transition).

```
public Test() throws
    java.rmi.RemoteException {
    super();
    setCurrentState("Defined");
    transitions = new Transition[12];
    for (int i = 0; i < 12; i++)
        transitions[i] = new Transition();
    transitions[0].initstate = "Active";
    transitions[0].finalstate = "Suspended";
    EventProfile ep0 = new EventProfile();
    ep0.fromString
        ("|EVENT#Suspend|false|");
    transitions[0].event = ep0;
    /* the rest of the method is omitted
       for brevity */
}
```

Two other methods are always present in any OPSS class entity. They are the `init` method that is in charge of managing

subscriptions to all events relevant for the evolution of the entity) and the `processEvent` method. This last one is called by the JEDI middleware when a new event has to be delivered to the entity. `processEvent` interprets the event and then calls the appropriate operation. The UML to OPSS translator exploits the knowledge of the information contained in state diagrams and in the activity diagrams to generate the code of such method:

```
protected void processEvent(JEDIEvent evt)
{
    super.processEvent(evt);
    if (evt.getAttributeValue("EVENT").
        equals("setResult"))
        setResult(evt);
    if (evt.getAttributeValue("EVENT").
        equals("ObjectCreated"))
        finalizeActivityTermination(evt);
}

protected void setResult(JEDIEvent evt) {
    JEDIEvent je;
    if (getCurrentState().toString().
        equalsTo("Active"))
        if (getResult(evt).equalsTo("True"))
            ErrorFound = true;
        else
            ErrorFound = false;
}
```

Most events are interpreted by the `processEvent` defined in the super-class. Two exceptions are events `setResult`, which (as specified in Figure 4) has to set `ErrorFound` (in method `setResult`), and `ObjectCreated`.

When an instance of activity `Test` completes, consistently with the process model shown in Figure 7, if `ErrorFound` is true (see the condition attached to decision D1 in Figure 7) a new instance of `ConsiderAR` is created, otherwise the process terminates. In both cases the instance of `Test` moves to the state `Complete`. Method `completeActivity`—generated by the translator (see below)—takes care of triggering such behavior. In particular, the code fragment that publishes event `CreateSuccActivity` triggers the creation of the new instance of `ConsiderAR`. Creation is performed asynchronously and causes event `ObjectCreated` to be issued. In turn, this event is received by the instance of `Test` which can finally move to state `Complete` (`processEvent` reacts to the event by terminating `Test` via method `finalizeActivityTermination`). Therefore, the termination of the activity is postponed after the receipt of `ObjectCreated` in order to guarantee that no race conditions occur. Notice that the state machine associated with the `Test` class does not mention events `CreateSuccActivity` and `ObjectCreated` since they are service events used by OPSS to manage the way activities are organized in a sequence.

```
protected void completeActivity(JEDIEvent
evt) {
    if (getCurrentState().toString().
        startsWith("Active") && (IsCompletable(evt)))
    {
        ReleaseArtifactResource(evt);
        if (ErrorFound == true) {
            EventProfile ep = new EventProfile();
            ep.fromString("|EVENT#ObjectCreated#
                CLASS#example.ConsiderAR#"+
```

```

    "NAME#ConsiderAR-1|false|");
    try {
        subscribe(ep);
    } catch(EDConnectionException ex){...}
    JEDIEvent je = new JEDIEvent();
    je.fromString("|EVENT#CreateSuccActivity
                #CLASS#example.ConsiderAR#...");
    try {
        dispatch(je);
    } catch (EDConnectionException ex){...}
    } else finalizeActivityTermination(evt);
}
}

```

The above example shows how the creation of new activities is delegated to the activity that precedes them in the process. This is always true in case of a single preceding activity. When we have multiple preceding activities (see the case of `CollectMRs` in Figure 7), the UML to OPSS translator creates a `joinMaster` class that is in charge of monitoring the termination of all preceding activities and of creating the new activity (or activities) when it is appropriate. For the sake of space we do not show the code associated with this class.

5.3 Implementation of the translator

UML process models can be written by means of any of the several available CASE tools supporting UML. It is clearly convenient to build a unique translator which is able to process the output of any modeling tool. Luckily, XMI [14] is emerging as the standard language for the textual representation of UML models. We therefore decided to build a translator which takes XMI (version 1.1.) files as input.

The general organization of the translator is depicted in Figure 9. The translation process is carried out in two steps. There are two main reasons for this choice. First, XMI is a very recent standard, not yet fully consolidated. In case XMI is changed, we will need to change just the first stage of the translation, while keeping the intermediate representation –and hence the second stage– unchanged. Second, the XMI files generally contain a great deal of information which is not relevant for describing the behavior of the process being modeled. The first step of the translation gets rid of this extra information. Meanwhile, it performs some useful processing, such as resolving references to type identifiers. The result is an equivalent yet much more compact representation of the model, which can then be processed much more efficiently. This compactness allows the intermediate representation to be checked more efficiently in order to assure that it contains all the information needed for a successful translation.

In the translation process we exploit XML standard parsers and translators. In particular, as shown in Figure 9, the first step is carried out by defining a set of fixed XSLT rules and by processing them by means of the Xalan XSLT processor. The result is an XML file whose structure is similar to that of the original XMI file, but conforms to a specific structure (specified by a DTD) we defined. This file contains:

- The classes defined in the class diagram, including the predefined OPSS model. Class definitions are complete with attributes, methods and methods parameters.
- Information reported in state diagrams (both predefined and user-defined).

- Information concerning the flow of the activities as it is reported in the activity diagrams.

The second step is made in a rather traditional way, by creating a DOM model of the data, and processing them ad-hoc for generating the corresponding Java classes.

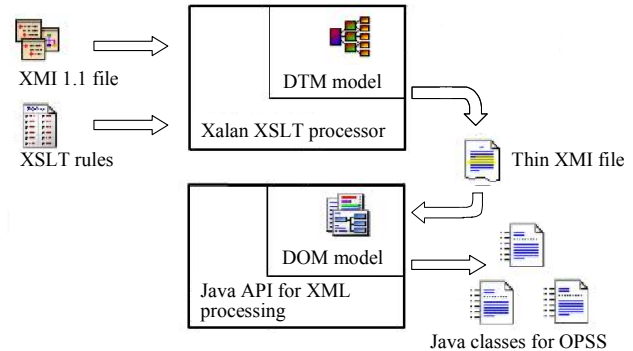


Figure 9 Structure of the translator.

6. A CASE STUDY

As a first test of the validity of the approach, we have modeled and enacted the process of managing anomalies detected during software testing and operation [2], [4]. When an anomaly is found, an Anomaly Report document is prepared and submitted to the manager responsible for the specific project. The manager is in charge of directing the procedures for anomaly analysis, error detection, and bug fixing. All these procedures are human-intensive: many of the activities to be executed cannot be automated, but require process agents' creative work. The following roles are involved in the process: the Project Manager (PM), responsible for the project; the Designers (DEs), responsible for the design and implementation of software modules; the Programmers (PRs), in charge of the coding activity.

Anomalies are usually detected during system test and operation. Each anomaly is described by an Anomaly Report (AR) stored under configuration management. When an AR is created, its initial state is "originated". PM considers ARs and decides how to handle them. Three alternatives are possible: the AR is "rejected", because it is not a real anomaly, or the AR is "approved", because it is relevant; otherwise, the AR is "postponed".

Each approved AR is passed to DEs to be analyzed. Each DE considers the module(s) she/he is responsible for and determines the necessary modifications that should be implemented (if any). For each proposed module modification she/he generates a modification request (MR) with initial state "originated". When all DEs have analysed the ARs, PM schedules a Configuration Control Board (CCB) meeting in order to decide which MRs have to be considered for the next release of the product. Each MR may be either "approved", i.e., the modifications have to be implemented for the next release; "postponed", i.e., the MR has to be reconsidered in the future; or "rejected", i.e., the proposed MR is not accepted. Once all MRs of an AR have been considered, the AR state is set to "defined". The modification proposed in a MR may be directly performed by the DE or it may be delegated to a PR. Upon termination, the MR state is set to "done". When all MRs associated with an AR are done, the AR state is set to "solved".

Note that the distribution of the AR to DEs represents a decomposition of the original activity (“Analyze AR”) into multiple sub-activities delegated to DEs. In this case, PM and DEs cooperate to accomplish the original “Analyze AR” task, by sharing a common resource, i.e., the AR. Such cooperation is accomplished asynchronously through e-mail or internal memos.

Conversely, some of the steps in the AR management process require group activities. For instance, during CCB meetings, PM and DEs interact to decide the release strategy. During the interaction they share access to the AR and the MRs.

6.1 Process model definition

Process model definition usually starts from sketching one or more activity diagrams describing the process flow. These lead to the definition of a set of proper classes and associated state diagrams. Finally, activity diagrams are properly refined and completed with all details. Figure 8 and Figure 10 show the class diagrams we have defined for our example. From class Artifact, we have derived classes AnomalyReport and ModificationReport that represent the documents managed during the process. We have defined three types of HumanAgent describing the three main roles in the anomaly management process. In order to assign the ownership of activity ControlMRs to the group of people composing the CCB meeting, we have defined CCB as a subclass of Group. The execution semantics associated to the activity owned by such a group is that it is executed only when all group participants are available for it.

In Figure 8 all the activities involved in the process are shown. They are all derived from the class HumanActivity. Notice the containment relationship between ConsiderAR and AnalyzeAR. It is used to represent a subactivity relationship. All other relationships between activities (precedence, parallel execution, sharing of artifacts, ...) are described in the main activity diagram of Figure 7. Figure 4 and Figure 5 show the state diagrams associated to some entities of the process.

6.2 Process model enactment

The process model described in Section 6.1 has been translated according to the rules described in Section 5. The resulting code is directly enactable by means of OPSS.

Figure 11 shows a snapshot of the process state when three activities of type AnalyzeAR have been instantiated. This situation occurs after the execution of an instance of ConsiderAR, as a consequence of fork F1. Notice that the number of AnalyzeAR instances depends on the number of designers (NumDesigner) participating in the process. Figure 12 illustrates the life cycle of activity instance ConsiderAR-1 at the same execution time. The box highlights the current state (Complete) of the activity instance. The attributes window shows the details of the activity state. A URL is specified containing a description of the activity to help the executor agent in his/her work. Also the name of the agent that executed the activity is provided. Since in the completed state the activity has

released any resource and artifact, the corresponding attributes do not contain any value.

Both Figures 11 and 12 are screenshots of the OPSS viewer user interface. As mentioned in Section 2, such component is used to monitor the process.

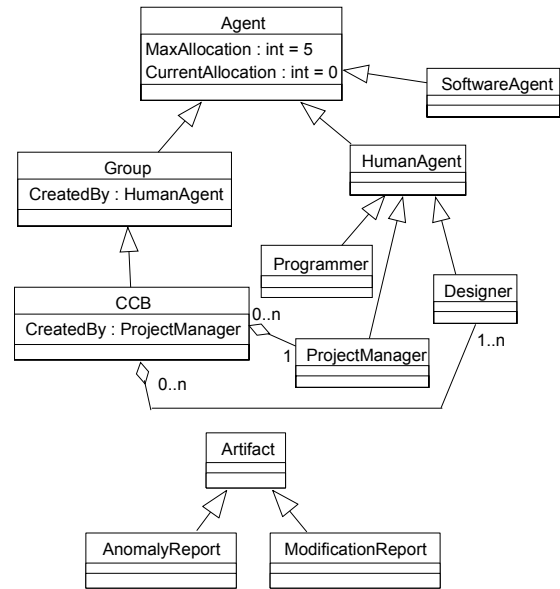


Figure 10 Class diagrams of agents and artifacts.

7. EVALUATION

In the software process research community it is well known that process modeling involves two different process representation issues: high-level, human-oriented representations, and low-level, detailed enactment-oriented descriptions. This opinion was neatly synthesized by Gruhn and Urhainczyk: “... it is above all important to find modeling languages which lead to an intuitive description. Those intuitive models should be refined step by step concerning the parts of the process that are well known and that demand for a low level description, in order to arrive at enactable models.” [9].

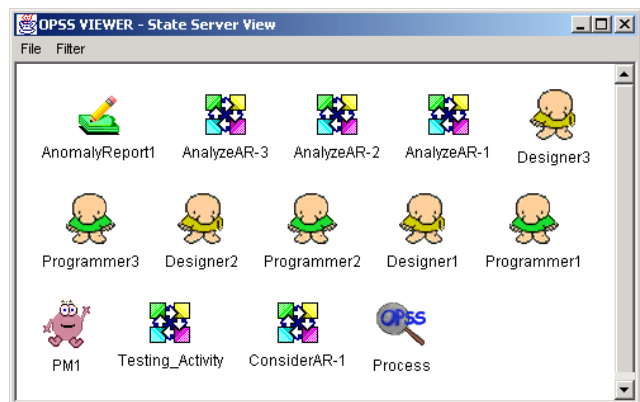


Figure 11 The OPSS monitor showing State Server entities.

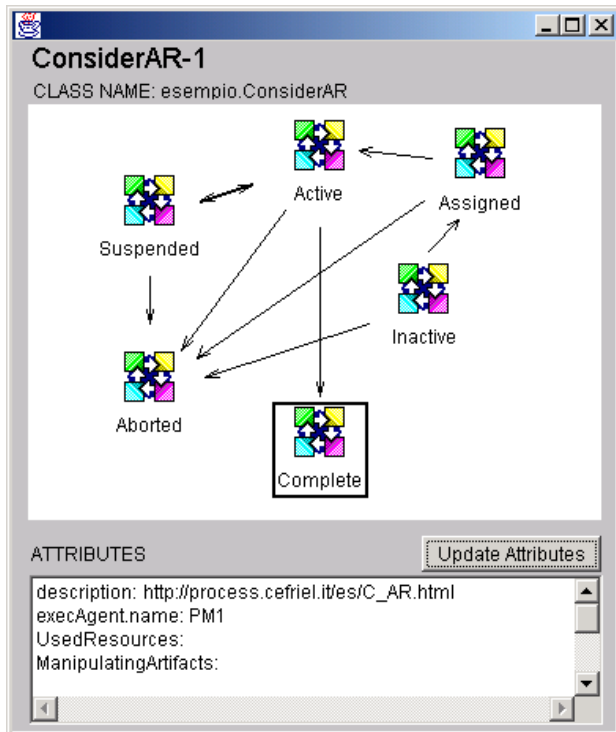


Figure 12 Activity states as shown by the OPSS monitor.

While a relevant number of PMLs have been defined, none of them has been completely successful in addressing at the same time the requirements of both high-level modeling and enactment. In SLANG [3] we have tried to address such requirements by identifying two levels of process representation: activity decomposition, concurrency, sequencing and synchronization among activities are modeled graphically, by means of high-level Petri Nets, while many other details such as association between activities, roles, and artifacts, mechanisms for interacting with the user, etc. are hidden in the implementation of the objects that are associated with tokens in the Petri net, and of the operations associated with transitions. Similar two-levels representations are also offered by several other PMLs, like –among others– Serendipity II [10] and Juliette [5].

Our modeling experiences [1], [2], [4] as well as others' [9] tend to confirm that it is not easy to integrate the linguistic tools needed for high-level and detailed modeling into one notation. In particular, we have noticed that in SLANG the representation of the same process varies depending on the purpose of the modeling activity. For instance, in [4] the goal was to provide the management of our industrial partner with a process model which could be used as a formal base for evaluating and discussing the properties of the process: as a consequence the model was a high-level one, with little details. In [2] the goal was to enact the process. Thus we had a very small number of simple SLANG nets describing activities and their interrelations, and a relatively large amount of code in order to provide the process engine with the necessary operative details.

According to our experience (and to others' [8], [11], [12], [13]) UML is definitely a good PML, which provides a set of linguistic

tools for modeling process structure, behavior, resources, constraints, etc. at different levels of abstraction.

However, little work has been done for making UML models enactable. We achieved the ability to write relatively high-level process models in UML, while making them precise enough via inheritance from model elements corresponding to OPSS classes. Automatic translation was made possible by the commonalities of the linguistic paradigms (object-orientation and state machines) adopted by UML and OPSS. The translation produces a programming-level description of the process that can be efficiently enacted by OPSS. Note that as long as many other process support systems employ object-orientated PMLs and state machines, we could build translators for other process centered environments.

Therefore we can conclude that an approach based on UML and an automatic translation to an enactable PLM overcomes the problems connected with the dual language representation of processes.

Of course, it is important to evaluate how efficient the proposed approach is. Although we have not yet performed formal experiments, we can make some preliminary evaluations based on the work carried out for the case study. Modeling the case study took one day (not including the time –which does not depend on the PML– required for analyzing the process). Considering that the translation time is negligible, the achieved performance is very good: experienced OPSS programmers estimate up to five days the time needed to develop from scratch the corresponding OPSS enactable model. The reason for such a good performance is that we worked at the model level, which is much more efficient than working at the code level. Moreover, we needed no time for debugging of the Java code, as long as the automatic translation guarantees correct code. Testing was devoted almost uniquely to prove the correctness of the process model, not the correctness of its implementation.

An additional evidence of the advantages of the approach can be found by comparing the process model in 6.1 with the one presented in [4] where we used SLANG to describe the same process. Even though in [4] we tried to use the language at a very high level of abstraction in order to provide a human readable model, we took much more time to work out the model and the result is much less readable to a profane and indeed not enactable.

As a final benefit of our approach, UML models can be easily packaged to be reused in the models of different processes.

The limits of the work done till now are the following:

- If a modeler does not provide all the necessary information in state diagrams, the code produced by the automatic translation will not be complete. However, the modeler is always free to edit the code produced by the translator, thus changing or adding the features of the process as desired.
- Up to now we did not consider modeling the interface with the human agents and/or the development tools used in the process (i.e., the elements appearing in the upper part of Figure 1). Sometimes it could be interesting to model in UML such interfaces (e.g., to describe the behavior of a tool, which messages it generates under what conditions, etc.). However this is not a central issue as long as tools are generally given, not developed specifically for the process. In any case the implementation of tools is quite different from

the enactment of processes: for tool development the traditional usage of UML is perfectly suited.

8. CONCLUSIONS

In this paper we have discussed the usage of UML as an *enactable* process modeling language. For this purpose we have exploited the basic constructs provided by standard UML, without introducing new stereotypes or other extensions. In fact, a disciplined usage of the available constructs and a few predefined classes corresponding to the core elements of process descriptions (activity, artifacts, resources, etc.) allowed us to model a fragment of a non-trivial process.

An important role in the presented work was played by the translator, which automatically converts UML process models into Java code. This enables the usage of a process-centered environment (namely OPSS) to enact the process.

The main lessons learned from the work presented here are that UML can be effectively employed to model enactable software processes. The automatic translation of models written in a high-level PML (i.e., UML) into a low-level description makes the usage of a process environment like OPSS possible to a larger community, and definitely more efficient.

Future work includes several issues. We are planning to employ our approach to model other processes like RUP. Also it would be interesting to translate UML models into other notations that can be executed by different process-centered environments. This would allow us to understand if the advantages we have found in using UML can be generalized to other contexts.

9. ACKNOWLEDGMENTS

The work described here was carried out as part of the ITEA project DESS (Software Development Process for Real-Time Embedded Software Systems). Project DESS is partly supported by MIUR.

10. REFERENCES

- [1] Arlow J., Bandinelli S., Emmerich W., and Lavazza L., Fine Grained Process Modeling: an Experiment at British Airways, Software Process Improvement and Practice, J. Wiley, 3,2, 1997.
- [2] Bandinelli S., Di Nitto E., and Fuggetta A., Supporting cooperation in the SPADE-1 environment, IEEE TSE, 22, 12, 1996.
- [3] Bandinelli S., Fuggetta A., and Ghezzi C., Software Process Model Evolution in the SPADE Environment, IEEE TSE Special Issue on Process Evolution, 19, 12, 1993.
- [4] Bandinelli S., Fuggetta A., Lavazza L., Loi M., and Pico G.P., Modeling and Improving an Industrial Software Process, IEEE TSE, 21, May 1995.
- [5] Cass A., Lerner B., McCall E., Osterweil L., Sutton S. Jr., and Wise A., Little-JIL/Juliette: A process Definition Language and Interpreter, in Proc. 22nd ICSE, Limerick, Ireland, June 2000.
- [6] Cugola G., Di Nitto E., and Fuggetta A., The JEDI Event-based Infrastructure and its Application to the Development of the OPSS WFMS, *IEEE Transactions on Software Engineering*, September 2001.
- [7] Cugola G., Di Nitto E., and Fuggetta A., Exploiting an event-based infrastructure to develop complex distributed systems, In the Proceedings of the 20th International Conference on Software Engineering (ICSE 98), Kyoto, Japan, April 1998.
- [8] Eriksson H.E., and Penker M., Business Modeling with UML, Wiley Computing Publishing, 2000.
- [9] Gruhn V. and Urhainczyk j., Software Process Modeling and Enactment: An Experience Report related to Problem Tracking in an Industrial Project, Proc. ICSE 1998, Kyoto.
- [10] Grundy J., Apperley M., Hosking J., and Mugridge W., A decentralized Architecture for Software Process Modeling and Enactment, IEEE Internet Computing, 2, 5, 1998.
- [11] Jacobson I., Booch G., and Rumbaugh J., The Unified Software Development Process, Addison-Wesley Object Technology Series, 1999.
- [12] Jager D., Schleicher A., and Westfechtel B., Using UML for Software Process Modeling, In Proc. of ESEC/FSE'99, Toulouse, France, LNCS 1687, Springer, September 1999.
- [13] Marshall C., Enterprise Modeling with UML: Designing Successful Software through Business Analysis, Addison-Wesley, 2000.
- [14] OMG, XML Metadata Interchange (XMI) Specification, Version 1.1, November 2000, <ftp://ftp.omg.org/pub/docs/formal/00-11-02.pdf>.
- [15] OMG, Unified Modeling Language Specification, Version 1.3, First edition March 2000, <ftp://ftp.omg.org/pub/docs/formal/00-03-01.pdf>.
- [16] Osterweil L., Software Processes Are Software Too. Proc. 9th ICSE., ACM Press, New York, N.Y., 1987, pp. 2-13.
- [17] Schiavoni M. and Trombetta M., Uso di UML come linguaggio di descrizione di workflow. Progettazione e sviluppo di un traduttore verso il sistema di workflow OPSS. Thesis, Politecnico di Milano, June 2001. (In Italian).
- [18] WFMC, Workflow Management Coalition Interface 1: process definition interchange process model, Technical Report WFMC-TC-1016-p, ver. 1.1, October 1999.