A FRAMEWORK FOR THE MODULAR DESIGN AND IMPLEMENTATION OF PROCESS-AWARE APPLICATIONS

Davide Rossi and Elisa Turrini

Dept. of Computer Science
University of Bologna
Mura Anteo Zamboni, 7
I-40127, Bologna, Italy
{rossi | turrini}@cs.unibo.it

ABSTRACT

Process-aware software systems are establishing themselves as prominent examples of distributed software infrastructures. Workflow Management Systems, web service orchestration platforms, Business Process Management support systems are relevant instances of process-aware software systems. These systems, because of their own nature, are all characterized by presenting a behavioral perspective, that is a perspective describing the steps that can be executed during the enactment of the process.

In this paper we present a framework for the design and implementation of the behavioral perspective in modular process-aware architectures. This approach can be applied across different application domains. The modularity of the resulting architectures is well-known as a key factor in achieving software qualities such as reliability, extensibility, robustness, maintainability and ease of use.

Our framework is based on EPML [21][22], an executable process modeling language, and on its enactment engine. EPML has been designed with the aim of promoting separation of concerns, easing the modular approach to the design of process-aware software architectures. A notable advantages of the presented approach is that of using the same modeling language and the same modular software component to support the behavioral perspective across different application domains. As an example we show how it is possible to use EPML and its engine in the modeling and the implementation of a wide spectrum of software architectures, from those supporting business process simulation to those supporting web service orchestration.

Keywords

Process-aware systems, Process modeling, EPML, Software engineering.

1. INTRODUCTION

Process-aware software systems are establishing themselves as prominent examples of coordination-based software infrastructures, following a trend that sees computer systems shifting their focus from data to processes. Workflow Management Systems (WfMSs), web service orchestration platforms, Business Process Management support systems are relevant instances of this class of systems. A modular approach to the design of architectures supporting process-aware systems is that based on perspectives (a concept introduced in [11] and later refined in [13] and [28]). A process can be characterized by different perspectives: the functional, describing what has to be executed; the organizational (or resource), describing who (software component or human) is in charge of the execution; the informational (or data), describing what data has to be processed and the behavioral (or process, or control-flow) describing when part of the process has to be executed during its enactment. In this work we focus on the latter perspective, by presenting a framework to address it, based on an effective separation of concerns. This promotes modularity and allows the design of process-aware architectures able to support a wide spectrum of applications. Our approach is based on EPML, a graphical, executable, process modeling language and on its enactment engine. As a result the very same notation can be used to model the process perspective in workflow systems, business process simulation systems, web service orchestration systems, process-aware web applications and other process-aware software systems. Moreover, the software architectures supporting these systems can be designed in a modular fashion, composing together application-domain specific components and the EPML engine.

This paper is structured as follows: section 2 outlines the main issues we tried to address in designing our proposal; section 3 introduces EPML and its enactment engine. The sections 4, 5, 6 and 7 show sample methods and architectures to design and implement process-aware applications using our modular EPML-based framework. Section 8 discusses our proposal with respect to related work. Section 9 concludes the paper.

2. COORDINATION FOR PROCESS-AWARE APPLICATIONS

The Babel of business process notations, workflow and web service orchestration languages keeps growing day by day. Most of these languages/notations have not been designed to provide improvements in the description of the dynamic behavior of process-aware applications but, rather, to address some domain-specific issue (like supporting web services invocation to coordinate an orchestration or managing resources in a workflow). This forces the members of the development team to learn different tools to address the same aspect: the behavioral perspective. Moreover, it may well be the case that in a single, large application, coordinating users connected to different software systems and remote components, the behavioral perspective has to be addressed with different languages/notations at the design level and with different enactment engines at the implementation level.

A partial solution to this problem is provided by BPMN [16]. BPMN is a graphic notation to model business processes that has not an associated formal semantics and that does not produce executable specifications. A BPMN diagram could, however, be transformed into different executable notation. BPMN's specifications, for example, suggest a mapping to transform a diagram into BPEL. From a conceptual point of view the same approach can be taken with respect to different "target" notations, providing a unified high-level modeling tool. This would imply that, at least at the design level, a single modeling notation can be used to be later transformed into one or more executable ones. As discussed in section 4, however, this solution suffers several limitations. It is our opinion then that, while BPMN is a highly valuable contribution for high level process modeling in early phases of the software development process and for documentation purposes, an executable notation (and possibly a single one used within one process) has clear advantages in the design-related activities.

These are the reasons that lead us to the development of a framework to address the behavioral perspective in different application domains using the same tools. This approach is based on EPML, an exogenous [7] coordination language that has been designed in order to obtain an effective separation of concerns, easing the modular approach to the design on process-aware software architectures. It should be noticed that EPML defines the interactions that can take place between the actors in the system, but cannot change them at run time like, for example, Manifold [8] and other control-driven [18] coordination languages can do. Process modeling languages, in fact, can be instances of different coordination models (control-driven, data-driven, space-based, rule-based and so on) and, while EPML's characteristics are shared with most existing workflow languages and process modeling languages (that are *flow languages* in the broad sense of languages that describe the process in term of, potentially concurrent, flows of executions, their interactions and their synchronizations) there are contexts in which different paradigms are more suitable and should be preferred.

EPLM is a graphical, executable language with a formal semantics and high level of expressiveness. It is our opinion that these characteristics are essential in this kind of tools. While some of the reasons for this are rather obvious (a diagram is easier to understand than a sequence of text lines) some are less immediately apparent. As an example of this consider the following issue: WSBPEL [6] is a textual (XML-based) web service orchestration language that has no graphical representation but most of the available tools that support WSBPEL development provide a graphical modeler based on a proprietary notation. The diagrams produced with these tools become artifacts of the software development process introducing non-standard notations in the process and causing potential vendor lock-in problems.

Details about EPML and its engine can be found in section 3; in this section we focus on how a process modeling language can be designed to maximize separation of concerns. Separation of concerns is a well-known topic in software engineering in general and has specific relevance in the research area related to coordination models and languages. Separation (orthogonality) between coordination and computation is a cornerstone for this area [12]. In a coordinated process the computation is carried out by software components or human beings (*actors*) participating in the process; the coordination is carried out by a coordination runtime. The distinction between coordination and computation however is, in our opinion, too coarse. Coordination should really be split in *interaction model* and *process logic*. The interaction model defines the execution flows, i.e. the possible interactions among the involved entities. The process logic defines which, among all possible execution flows, have to be activated. To show the relevance of the process logic and the fact that its importance is often neglected we use BPEL again as a paradigmatic example (in this article we use BPEL to refer to either BPEL4WS [3] or WSBPEL, the two versions of the language). BPEL uses XPath expressions as predicates to support control flow decisions; there are cases in which, however, these decisions imply a computational effort for which XPath is not well suited. In these cases the decisions are delegated to activities that have to be created ad hoc for this task; as a consequence two different perspectives get mixed together (besides the impact on software qualities such a solution also implies that new web services have to be created and hosted somewhere in order to support the behavioral perspective). Implicitly acknowledging this limitation all existing commercial BPEL engine implementations we are aware of, include custom extensions that allow to activate software components implementing the process logic. The latest version of BPEL, WSBPEL, also provides a standardized extension mechanism for this. For the same reason, BEA and IBM (some of the partners that supported the development of BPEL) proposed an extension of BPEL4WS: BPEL-J [2]. In BPEL-J, Java code *snipets* can be embedded in a BPEL specification reducing the need to delegate to external components. The main drawback of this approach is that the process logic can be expressed using a specific language only. This implies that only users proficient in Java can benefit from BPEL-J; moreover, from a technological point of view, workflow engines must be able to activate Java programs.

To better support separation of concerns, in EPML the computation, the interaction model and the process logic are addressed by distinct elements, that is, respectively, *activities*, *processors* and *processors' logic*. The first two are (graphical) elements of the language; processors' logic is a program fragment (expressed with any suitable language) supporting the decisions associated to processors. This clear distinction between these three aspects is propaedeutical in promoting a high level of modularity allowing us to use the framework based on EPML to support the behavioral perspective in several different application domains as we show in this paper.

Another aspect which is essential for the wide applicability of our framework is related to the expressive power of EPML. The expressive power of workflow languages has been subject to several investigations in the last few years, these studies can be (and have been) applied to process modeling languages as well. The main problem addressed in this research field is the fact that there is not a formal metric to evaluate the expressive power. One of the most successful approaches is the one based on the workflow patterns [5]; this is an analysis strategy that evaluates the languages with respect to their ability to model a set of predefined (sub)processes. Expressive power is a critical parameter to increase the suitability of a process modeling language (for the quite obvious fact that a language that cannot easily model the interactions within a process surely poses large limitations to its own usage). Most existing workflow languages, for example, cannot easily model a large number of common real-world interactions. A workflow pattern-based analysis shows that EPML supports all the 20 patterns described in [5] (with the partial support of implicit termination because of a design decision); an extended pattern set has been presented in [25], EPML support most of these 42 patterns with minor exceptions (like interleaved parallel routing and critical section); extending the semantics of the language to capture all patterns is possible but that would make the language itself more complex so, at this time, we decided to not support them. In the design of EPML, in fact, we addressed the workflow patterns but we also strive to achieve a good compromise between simplicity and power.

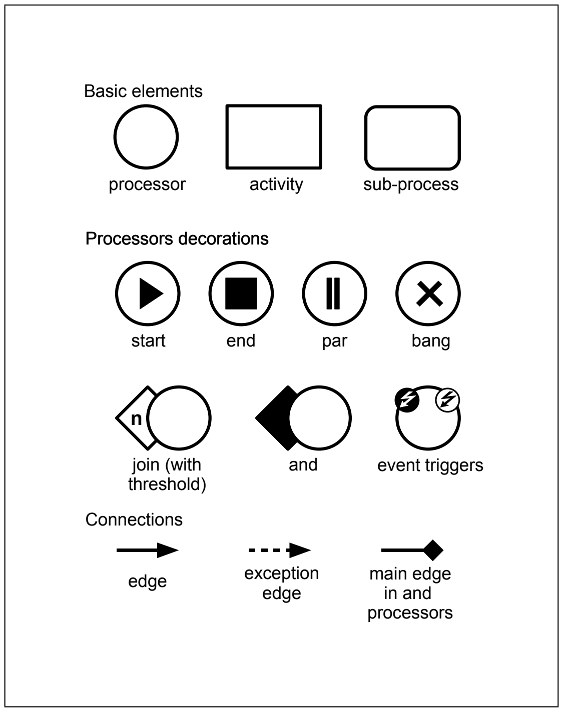


Fig. 1. The components of a EPML diagram

3. EPML

EPML is a graphical process modeling language that enables the representation of a process interaction model using a directed graph in which oriented edges are used to define the execution flow structure.

In this paper we call EPML *specification* a process modeled with EPML. The specification can be a *diagram* or an XML document. It is possible to translate an EPML diagram in its XML representation and vice versa. A process specification (in form of XML document) can be executed by an *engine*; we refer to a specification in execution as a *process instance*.

The purpose of this section is not to provide a detailed description of EPML (interested readers can refers to [21]) but only to briefly introduce its main features.

The components of EPML are shown in Fig 1.

Two types of nodes exist in EPML: *activities* (represented with squares) and *processors* (represented with circles); subprocesses (represented with rounded rectangles) really are just folding of subgraph with specific characteristics. Activities are elements of computation: they can be either external applications or work items allocated to a workflow participant (possibly a human actor). An activity can have an entry edge, an exit edge, and, potentially, an exception edge. Processors are elements of coordination: they implement the process logic possibly using a standard programming language. A processor is like a gateway: it can perform synchronizations and uses the process logic to perform routing decisions; as such it can have multiple entry edges and multiple exit edges.

Each execution flow is represented by a *token*. A token contains the *data* associated to the flow and produced and/or used by activities and processors. Moreover tokens also contain identifiers that allow to implement flows synchronization. Similarly to other formalisms (e.g. place-transition networks), the set of tokens present in a given moment, and their location in the diagram, represent the state of the system.

Each time a flow (i.e. a token) reaches a node, and its associated entry condition, if present, has been satisfied, the node *activates* (a new *node instance* is created). Concurrent activations of the same node are possible inside the same process.

In EPML, two types of edges exist : the *standard edges* and the *exception edges*. The latter are activated only in the case of anomalies (an exception raised by an activity or by an unsatisfiable join associated to a processor). Activating an edge means sending a flow over it, (i.e. putting the token representing the flow into the edge's destination node).

A processor can be decorated with the decorations shown in Fig. 1; the entry decorations (and, join, join-with-threshold) and the exit decorations (bang, par) can be used for representing control-flow operations like join, split and sub-flow creation.

In the following paragraphs we briefly describe the semantics of every decoration that can be associated to the processor nodes. The decoration(s) a processor supports characterize(s) the type, and then the behavior, of the processor itself.

The *start* processor and the *end* processor (obtained decorating a processor with a *start* and *end* decoration, respectively) represent the start point and the end point of a process.

A processor with no decorations is a *simple processor*. It manipulates the data received by the previous node(s) in order to select the exit edge on which the execution flow will be routed.

A processor with a bang decoration (*bang processor*) enables to split a single process execution flow in parallel flows, and route them on one or more exit edges. The exit edges of a par processor can be labeled with a cardinality notation (à la UML) indicating the minimum and/or maximum number of process flows that can be routed in parallel

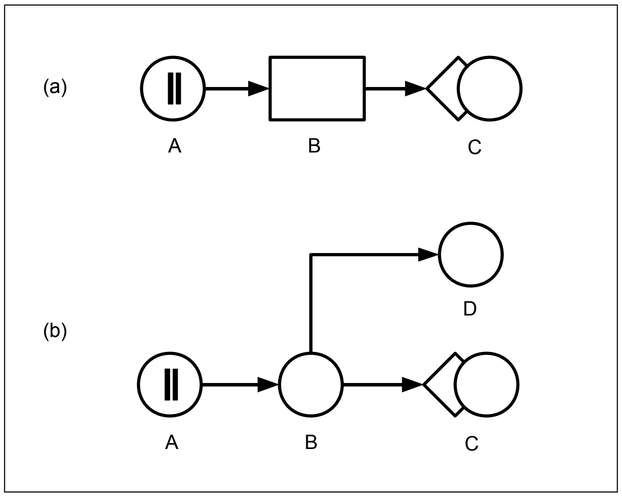


Fig. 2. Examples of how par and join processors can be composed

on the edge(s). If such a notation is not present, no limitation about the number of flows is imposed.

A processor with a par decoration (*par processor*) has a behavior which is similar to that of a bang processor, with the difference that the flows produced by this processors are sub-flows of the entry flow, that can be later synchronized. Technically this is accomplished by extending the token's identifier (which is a sequence) with new unique elements shared among all the related sub-flows.

The goal of a processor decorated with a join decoration (*join processor*) is to synchronize process flows (generated by one or more par processors) that are executing in parallel. The join semantics in EPML is quite sophisticated and it significantly contributes to make powerful the EPML expressiveness. For this reason, we present the peculiarities of the join semantics and discuss some examples. Intuitively, we can say that a join processor activates when all flows produced by the same instance of a par processor reach it, or when at least one flow is arrived and the other flows produced by the same par processor instance cannot reach it any more (for example because they have been canceled or they have been routed elsewhere). The join processor removes from the flow(s) the identifiers that have been used to implement the synchronization and merges the sub-flows in a unique flow.

The first example we discuss is shown in Fig. 2*a*. In this graph, when the flow reaches the par processor *A*, an instance of it is created and the associated process logic is executed. Let us suppose that this instance generates two flows marked with the same identifier, e.g. A_1. Whenever one of these flows reaches the activity *B*, a new activity instance is created. When an activity instance completes, the flow is routed on the exit edge (notice that the termination order of the activities instances can be different from the activation order). The processor *C* activates when it receives both flows marked with A_1 (that is when both executions of B complete).

It is important to remark that the synchronization operated by the join processor is related to the sub-flows generated by the same par processor instance. For example, let us suppose the par processor *A* activates twice; the first time it produces two flows (with identifiers A_1), while the second time it produces three flows (with identifiers A_2). In this case the join processor activates both when it receives the two flows marked with A_1 and when it receives the three flows marked with A_2 (again, notice that these flow can interleave in different ways).

We would like to point out that, according to the join semantics, a join decoration is satisfied even in the case at least one flow has reached the join processor and the other flows produced by the same par processor instance cannot reach the join processor any more. Cases like that can occur, for example, in a process fragment like the one depicted in Fig. 2*b*. Let us suppose that the par processor *A* generates two flows marked with the same identifier, e.g. A_1. Both flows arrive in the simple processor *B*, then it activates twice. Let us suppose that one of the flow is then redirect to the join processor *C*. This processor receives the flow but it cannot activate, since another flow marked with A_1 (that at the moment is in *B*) can reach it. In this situation *B* could:

- route the flow on the edge that conduces to the processor *D*;
- route the flow on the edge that conduces to the join processor *C*;

In the first case, the join processor *C* can activate as soon as the flow is sent on the edge; In the second case the join processor *C* can activate as soon as it receives the flow. In both cases, indeed, the activation of the join processor *C* is possible as no other flows generated by the same par processor instance can reach the processor anymore.

In a EPML graph, nodes can be connected in a free structure; this implies that, in general, there is not a 1-to-1 relation between a par processor and a join processor. A join processor can then synchronize flows coming from different par processors. To this end, a join processor maintains a list of par processors it refers to (we named this list *par set*) and activates only when all process flows generated by the par processor present in the par set reach the join processor. The par sets are calculated by means of an algorithm based on a network coloring mechanism (for sake of conciseness the algorithm is not described in this paper). This algorithm runs before the process specification starts to execute; the par set of a node is the same for every process instance and does not vary during the process execution. The rationale of this algorithm is that the par set of each join processor must contain all par processors that are directly connected to it (in the sense that there is a path between the two with no other join processor in the between) or indirectly connected (in the sense that there is a path between the two in which each par processor has a complementary join processor later in the path).

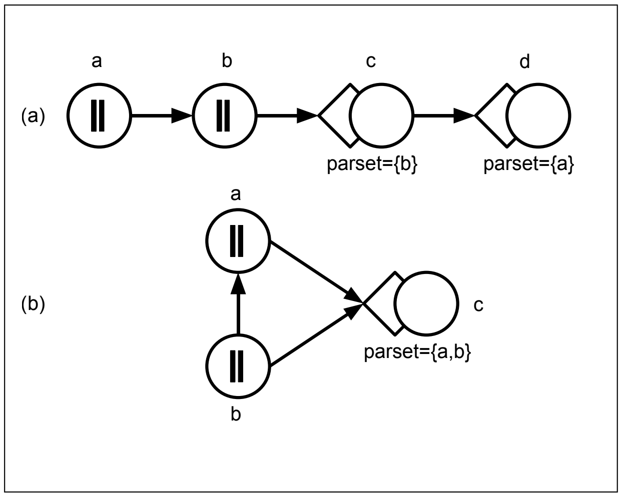


Fig. 3. Examples of structured and unstructured par-join composition

In Fig. 3 are shown two examples of structured and unstructured par-join composition. In Fig. 3*a* the parset of the join processor *C* contains the par processor *B*, and the parset of the join processor *D* contains the par processor *A*. This means that *C* synchronizes the flows produced by *B* while *D* synchronizes the flows produced by *A*. In Fig. 3*b* the par set of the join processor *C* contains both the par processors *A* and *B*. This means that *C* synchronizes the flows produced by both *A* and *B*.

A process decorated with a *join-with-threshold* decoration (*join-with-threshold processor*) is a special type of join processor. The threshold (i.e. the number n written inside the decoration) indicates the number of process flows the join processor has to wait before activating. The threshold can be positive, negative or zero. If it is positive, the processor waits for n flows; otherwise it waits for the number of generated flows minus $|n|$. Note that, if the threshold is zero, the joinWT processor waits for all the flows produced by the par processor instance. When the joinWT decoration can not be satisfied (i.e. no other flow can reach the joinWT processor), the joinWT does not activate and the flow is routed on the exception edge (if present).

A processor decorated with an *and* decoration (*and processor*) implements a different kind of synchronization. The and processor must have a main entry edge and can activate only when the following conditions are satisfied: (1) flows must arrive in all its entry edges; (2) incoming flows must be sub-flows of the flow coming from the main edge. Only the flow coming from the main edge is driven forward, the other flows are simply discarded.

A processor decorated with an *event* decoration (*event processor*) activates only if the entry decoration (if present) has been satisfied and an event is arrived. Events can be permanent or transient. Permanent event can be stored and used subsequently; on the contrary transient events are lost if the processor can not be activated at the moment the event arrived.

EPML supports *subprocesses*. A subprocess is just folding of subgraph with specific characteristics. When an execution flow enters in a subprocess, a new subprocess instance is created; the entering flow is considered as a new flow, that is all identifiers associated to that flow are ignored (they will return valid only when the flow exits from the subprocess). Processors inside a subprocess can add identifiers to the flow, however those identifiers are valid only inside the subprocess instance that created them, and are canceled when the flow exits from the subprocess.

EPML has a cancellation construct that enables to define a *cancellation area*. A cancellation area is a set of nodes and can be graphically depicted with a dashed perimeter connected to a node. When the node is activated, the flows and the node instances inside the cancellation area are removed or forced to terminate, respectively (in general, the flow that activates the cancellation and the canceled flows have to be generated by a common par processor instance).

EPML also makes available language elements that are just syntactic sugar, this is the case for the messaging (or asynchronous) activities and for timed activities and subprocesses. Messaging activities are depicted like activities decorated with the same glyph used to represent events in event triggers. Despite their graphical appearance, these really are processors whose process logic generates events (by interfacing with the enactment engine run-time system). When these events are used to communicate with components that have to be coordinated the usage of an activity-like element remarks the interaction with the functional perspective. A timed activity is an activity for which a deadline is set as soon as its execution starts. If the execution does not end before the deadline the activity is canceled. An exception edge can exit from a timed activities; when a time-out occurs the entry flow is routed on this edge. Being syntactic sugar a timed activity is internally expanded into regular EPML elements that support the aforementioned semantics. Timed sub-processes behave in a similar fashion.

As hinted above, EPML also includes elements related to the informational perspective. In this perspective we find two classes of data: production data and control data. Production data comprise all data that are essential for an application area. Control data can be either information related to the internal state of a process or pointers to production data (allowing process elements to access the information that has to be processed). In EPML a *data bag* (an associative map) associated to each token is used to manage the control data. By using special shared and transient entries, EPML provides a simple way to manage global data (that is data that has to be shared among all the processes hosted by an engine); process data (data that has to be shared by all the instances of a specific process); instance data (data that has to be shared within a single process or subprocess); and transient data (data that is automatically removed after the execution of an activity). For example process data has to be set (and retrieved) using the `$process` entry automatically added to the data bag of all generated tokens. This entry is a new associative map shared among all the instances of a specific process.

A formal semantics for EPML, based on a transition system, is described in [21] (although it is not updated to the latest version of the language). A formal semantics can be the starting point for the formal verification of specific properties (reachability, liveness, etc.). It is our opinion, however, that this argument is of less importance with respect to the ability to define the behavior of the language with no ambiguities. Anyone that had the chance of working with these languages to model complex processes went through the "try and see what happens" approach: when interactions get complex, the only way to be sure about the actual behavior of the system is to enact the process in a testing environment (and, quite frequently, the observed behavior is not the one expected by reading the manuals).

3.1. The EPML engine

A process modeled with EPML can be enacted by means of a software component called EPML engine; it takes as input an XML representation of the EPML specification and it executes it. The EPML engine is a software component written in Java 1.5 (and thus portable to most platforms). It has been designed as an event-based architecture: it consumes events and produces both events and state transitions. An event can represent an external event, the termination or the activation of a node instance. A state transition can entail modifying tokens or moving them in the network.

In many situations, the engine is expected to interact with human actors and/or software components. This implies the integration with a software architecture designed for a specific application domain. This integration has been implemented managing the events that the engine produces (*output events*) and generating the events that the engine consumes (*input events*). The input events can be *start events*, *external events*, and *end-activity events*. The start event triggers the execution of a new process instance; the external events are those caught by an event processor and typically are generated by the environment in which the process is enacted, and the end-activity events notify to the engine that an activity has terminated its execution. The output events can be *termination-process events* or *start-activity event*. They notify the external component that the process is terminated or that a new activity instance has been created and can start its execution.

In our system, the event notification is achieved exploiting the *implicit invocation* principle. To notify an input event to the engine the software architecture invokes an engine method that adds the event to its input-event queue; it will be then examined and proper actions (e.g. processor activation) will be taken. To notify an exit event to an external component the engine invokes a method on it.

The activities are not part of the engine, but they are external components that interact with the engine by means of a Java class that extends an `ActivityWrapper`. This class has a method (named `run`) that is invoked by the engine to start the activity execution. The run method contains the code that enacts the activity execution. Such execution can imply, for example, either the invocation of an external software component or the addition of a task into an actor task list. When the activity has terminated its execution, the `ActivityWrapper` produces a termination-activity event and passes it to the engine.

The architecture just described is very flexible as it enables the interaction among most kinds of external components. The engine integration in an existing architecture does not require any modification in the engine itself, but can be achieved by extending the `ActivityWrapper` class and putting into it the code that enables the interaction with the external component.

The activities execution is coordinated by the processors and specifically by the process logic. Technically speaking, the process logic can manipulate both production and control data. In some context, the process logic could also be required to decide which actor(s) the activity(-ies) has(have) to be assigned to. Such feature is typically related to the *resource perspective* in a workflow system. It can be integrated in EPML in a way transparent to the engine, that is specifying, in the token's data bag, which actor is in charge of executing the activity.

The process logic has to be specified by means of a suitable formalism. This could be a generic programming language, a scripting language or a language based on XML (e.g. XQuery or XSLT), provided that specific adapters implementing proper bindings are provided to interface to the engine. It is also possible to specify the name of a Java class implementing the process logic. The engine makes available to the process logic mechanisms for manipulating the tokens' data bags; the process logic has to inform the engine about which exit edge(s) has(have) to be activated.

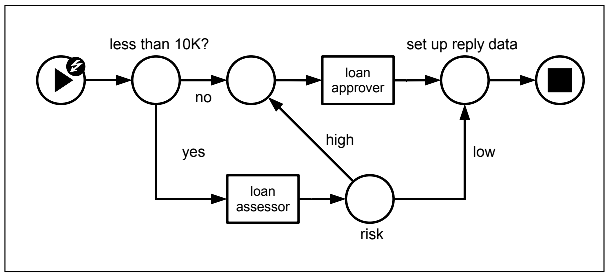


Fig. 4. A loan process

4. EPML.WS

EPML.WS is a software architecture for web services orchestration based on EPML. In order to support this specific applications class EPML.WS complements EPML with components that address the interaction with synchronous and asynchronous web services and allow a EPML-enacted process to be accessed as a web service.

EPML.WS is designed to be hosted inside a JEE architecture supporting JSR 109 (Implementing Enterprise Web services) and JSR 181 (Web Services Metadata for the Java Platform).

A very thin layer of software adapters have been provided in order to create the EPML.WS architecture. This includes:

- an adapter to transform incoming web services invocations (SOAP messages) into external events that are feed to the EPML engine;
- an activity wrapper (`ActivityWSWrapper`) to implement activities execution as web services invocation;
- a process logic wrapper (`WSXQueryLogic`) to use XQuery for routing decisions and for process data management.

Web services that have to be orchestrated are mapped into activities and the data perspective is managed using the simple integrated data perspective of EPML (that is using the data bags associated to the tokens). As usual, with EPML, the logic of the processor can be implemented with several languages. Given the fact that the data model used by web services is based on XML we provided a specific process logic adapter that allows to use XQuery (`WSXQueryLogic`). This adapter first creates the XML document against which the provided XQuery expression has to be run, it executes the XQuery expression and then parses the generated output to activate one or more exit edges and to modify the token's data bag (used to store process-related data and parameters for subsequent web service invocations).

We now use an example to give a basic idea of how to set up a web services orchestration with EPML.WS.

Consider the process depicted in Fig. 4. The activities `loan approver` and `loan assessor` are external web services that participate in a basic loan approval process. The process is initiated with the event associated to the start processor; in our case this event has been mapped to an incoming web service invocation by using the aforementioned adapter. The `less than 10k` processor has two duties: implement a routing decision (activating the outgoing `yes` or `no` edges) and prepare the parameters needed in the invocation of the two subsequent web services (associated to the `loan assessor` and the `loan approver` activities). Specific entries of the tokens' data bag are used to store incoming SOAP request messages and to set up invocation parameters. `less than 10k` has then to access the data associated to the SOAP request received at the beginning of the process to implement the routing decision and has to prepare the parameters for the forthcoming invocations. `less than 10k` has been set up to use `WSXQueryLogic` and it uses the same XML document associated to the original incoming SOAP request for the XQuery (element properties in the XML process description are used to specify how the XML document has to be produced from specific entries in the token's data bag; in this case a special `$wsin` elements containing the SOAP message of the incoming call). The XQuery expression, in this case, is as follows:

```
<xq_result>
  <edges>
  {
    if(data(//amount) > 10000)
    then <edge>yes</edge>
    else
      <edge>no</edge>
      <wsout>
        <param name="name">
          {data(//name)}</param>
        <param name="first name">
          {data(//firstName)}</param>
        <param name="amount">
          {data(//amount)}</param>
    </wsout>
  }
  </edges>
  <data>
    <entry name="name">
      {data(//name)}</entry>
    <entry name="first name">
      {data(//firstName)}</entry>
    <entry name="amount">
      {data(//amount)}</entry>
  </data>
</xq_result>
```

Once the query has been executed `WSXQueryLogic` parses the result in order to extract the information about the outgoing edge that has to be activated (this information is in an `edge` element), stores in the token's data bag all the name-value pair contained in `data` and sets up a special entry, `$wsout` in the aforementioned data bag (that is used to set up the parameters of the invocation of either `load approver` or `loan assessor`), by analyzing the `wsout` element.

If we assume that the body of the original SOAP request is as follows (namespace references have been omitted for clarity):

```
<body>
  <firstName>John</firstName>
  <name>Doe</name>
  <amount>1000</amount>
</body>
```

the previous XQuery expression produces:

```
<xqresult>
  <edges>
    <edge>no</edge>
  </edges>
  <data>
    <entry name="name">Doe</entry>
    <entry name="first name">John<entry>
    <entry name="amount">1000</entry>
  <data>
  <wsout>
    <param name="name">Doe</param>
    <param name="first name">John<param>
    <param name="amount">1000</param>
  </wsout>
</xqresult>
```

The information in this XML fragment is then processed as described before, activating the `no` outgoing edge and putting the name, first name and amount data both in the data bag (these are used later by the `reply` activity) and in the special entry used to set up the invocation parameters for `loan approver`. Before the process completes, the `set up reply data` processor, linked to the end processor, sets up (in the data bag) a special entry that is used by the web service adapter to create a reply message to the original process invocation.

EPML.WS poses itself as an alternative to systems based on WSBPEL. With respect to these systems EPML.WS has the following advantages:

- graphical notation;
- higher expressive power;
- ability to implement complex process logics without relying on external components.

It should be noticed that WSBPEL has not a "native" graphical notation but there are specific guidelines about the usage of BPMN to model BPEL processes. From a practical point of view, however, the mapping of a process modeling language into another is a very complex issue. The details for the mapping of BPMN into BPEL, for example, are largely incomplete and it is well possible to create a BPMN diagram for which the translation into BPEL is undetermined. Even when using techniques which are more advanced with respect to those presented in BPMN's specifications, the resulting BPEL code (when it possible to obtain it) is often a bad example of "spaghetti BPEL", which does not only pose a readability problem, it also creates monitoring problems (at which conceptual point of the process is the program executing this piece of spaghetti BPEL?) and modification tracking problem (modifications in the resulting BPEL code can hardly be reported into the BPMN diagram). For an in-depth analysis of this problem the interested reader can refer to [17]. These kind of problems are not only limited to the BPMN to BPEL mapping, but are usually found whenever a transformation between two process modeling languages have to be performed. It is easy to realize this is the case considering that most languages have different expressiveness (as defined in [5]), so most of the times a transformation has to map interaction patterns that are easily supported in the original notation and not easily (if at all) available in the target one.

To test EPML.WS against BPEL we set up a simple experiment. We used the very same process described above (which is a sample from the ActiveBPEL [1] distribution) we "unplugged" the BPEL engine and we replaced it with EPML.WS obtaining a working orchestrated process presenting the very same behavior and the very same (web service-based) interface. Another interesting part of this experiment was the analysis the effort needed to setup a BPEL-based solution and the EPML.WS one. While we did not run formal tests to access a vague parameter like "effort" our experience with computer science students shows that the same task (modeling a process of average complexity) can be accomplished using EPML.WS in about one tenth of the time with respect to BPEL, factoring out the time needed to address the link relationship problems (that are addressed in a very complex, yet powerful, way by BPEL and are not addressed at this time by EPML.WS).

EPML.WS has been mainly designed as a proof-of-concept. A working prototype has been implemented but it still has a few limitations (for example bindings have to be programmed by hand). Nevertheless EPML.WS is the proof that it is possible (and relatively easy) to design, and implement, a software architecture for a specific application domain in which advanced coordination mechanisms are required to govern interactions among distributed actors.

5. EGO

EGO (E-Game Orchestration) [20] is a software platform to deliver e-learning games based on EPML. EGO allows multiple users to be engaged in collaborative or competitive games by using a web-based interface. With EGO, games with various interaction patterns among the actors can be modeled, such as the ones occurring in turn-based and concurrent games (with or without synchronization steps). Given the large amount of possible interactions that can take place among actors, games are good candidates as case studies to test coordination models. One of the basic concepts in EGO's game modeling is the interface (in the sense of user interface). In EGO an interface is an activity assigned to an actor. The idea is that a game can be assimilated to a workflow system in which the interaction of players with their gaming interface corresponds to the execution of work items assigned to actors. By making their moves (using the interface) the players accomplish the work items. One of the main differences of this system with respect to classical workflow systems is that a single interface is usually assigned to the actors (otherwise a player might have to interact with multiple user interfaces to participate in the game). This problem can be solved with two different approaches. The first one is to explicitly cancel the activities that have been assigned (and not accomplished) to an actor right before assigning a new one. This solution entails no modification to the semantics of standard workflow languages but the specification becomes soon cluttered with cancellations. A second solution, the one we adopted, is letting the upper layer in the software architecture to take care of notifying the engine that the an actor finishes a previously assigned work item whenever a new work item is assigned to him by the engine. Please notice that while this solution can be perceived as a violation of the semantics of EPML this is not actually the case. The semantics that is violated is that of a workflow system, something that EPML is not. In EPML, in fact, the resource perspective is out of scope, since the focus is in the process perspective. Tokens' data bags can be used to transfer information related to the resource perspective, but this concept is not intrinsic to EPML.

As stated above EGO is a web-based platform and it has been designed within the JEE framework. Its structure is quite simple: the engine interacts with a web application hosted in a JEE servlet container. The interaction takes place by means of input and output events. The web application is composed by two servlets (`Dispatcher` and `Process`) and a software component that captures the start-activity events produced by the engine and maintains the association between actors and activities. The `Dispatcher` servlet queries this software component in order to obtain the activity that has to be assigned to an actor and then dispatches the associated interface to the player. When a player

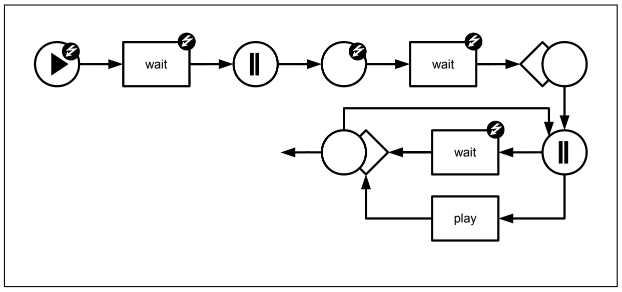


Fig. 5. The beginning of a simple turn-based game

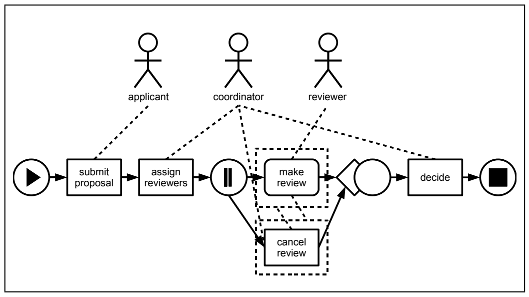


Fig. 6. A review process in EPML

submits its move, the move is processed by the `Process` servlet that produces an end-activity event embedding into it the data coming from the request sent by the player's browser. The event produced is then notified to the engine.

In Fig. 5 the first steps of a turn-based games are modeled. One player have a master role and is in charge of setting up the game environment for the other players. This player starts the game sending a message that triggers the creation of a new process instance. Other players join by sending a message that triggers the `new player` processor. The `wait` asynchronous activities are used to notify the players they have to wait for the other players to join before the game can proceed. Asynchronous activities are used to dispatch web pages for which no user input has to be reported back to the process. Once all the players have joined the game the turns can begin. During the turns one player is associated to the `play` activity (which is not an asynchronous activity, since the result of the play has to be reported to the process) while all the others players receive the `wait` interface. Once the play has been performed a new turn can start or the game can proceed with other phases.

EGO has been used to model several games and it is being currently used for e-learning purposes (with business simulation games). We also developed a version that is able to interact with AJAX-based presentation technologies.

6. PROCESS-AWARE WEB APPLICATIONS WITH EPML.WEB

EPML.WEB [23][24] is a platform (a model and a reference architecture) for the design and development of process-aware web applications based on EPML. The use of EPML in the context of process-aware web applications is motivated by the consideration that such applications should be able to support not just simple navigation and data access activities but (potentially complex) business processes. In this respect, we propose to model the business process with EPML and extend the engine architecture with software components that interact with the web navigational structure of the web application.

While navigating in the web application, users can access pages that are not related to any process or can visit process-related pages. We refer to the former as *standard navigation mode* and to the latter as *process flow mode*. To enter process flow mode the users follow specific *process-aware* hyperlinks. Process-aware hyperlinks can be used to create a (sub)process, to resume a previously started process that has been left (probably by using navigation links that drove the user outside the process flow), or to join an existing process created by another actor. In process flow mode, the sequence of the pages that are dispatched to the users may depend on the control flow of the process rather than on the navigation structure of the web application. Specifically, in our approach, the pages in process flow mode correspond to the tasks assigned to the users by the process.

Consider the process depicted in Fig. 6: it is a high level model of a simple project grant review process. In this diagram, standard EPML elements have been enriched with stick figures in order to model an elementary resource perspective. Applicants submit their projects for review, the coordinator assigns the actual reviews to a given number of reviewers, the reviewers make the reviews. A review is actually composed by two steps: a first evaluation is given considering an anonymous subset of the documents in the proposal, a second evaluation is given considering all the information related to the project, including the applicants' identity. While waiting for the reviewers to complete their work, the coordinator can decide to cancel a review (because it is delaying the process or for other reasons). When all the reviews have either been completed or canceled, the coordinator decides to reject or to fund the project. Please notice that the modeled process is a highly simplified version of what actually takes place in the real word. EPML has been designed with an high expressive power right because the authors acknowledge that real world (business) processes are far more complex than what academic papers seem to suggest. Nevertheless, given the focus of this work, an oversimplified example is reasonable.

To decide if a proposal should be funded or rejected, coordinators have to join the flow of the process generated by

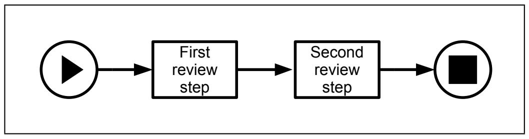


Fig. 7. The make review sub-process

the applicants when they submitted a proposal. In order to do that from within the web application, coordinators have to follow a process-aware hyperlink that drives them to a page related to the `decide` activity. This page should be available only when the `decide` activity has been assigned to the coordinator and it is a reasonable design strategy to prevent the generation of process-aware hyperlinks related to processes for which no activity is assigned to the current user. In the specific case of the aforementioned example, the `decide` hyperlink should be available from the proposal details page only when all the reviews have either been received or canceled. In general only process-aware hyperlinks used to start a new process can be always available. Even hyperlinks used to start sub-processes are not generally available (in the example, the make review sub-process can be started by reviewers only when they have been assigned the review of a proposal).

Consider now Fig. 7: it depicts the make review sub-process. As discussed above this sub-process is composed by two steps and it is very well possible that reviewers leave the sub process when finished the first step, before completing the second. As a consequence the `make review` hyperlink that is available from the pending reviews page can lead reviewers to the forms associated to the first or to the second review step depending on the state of the make review sub-process for that specific proposal.

Note that the web application framework must be able to interface to the process enactment engine in order to query the activities assigned to specific users and to signal the completion of activities. From the modeling point of view, the following issues have to be addressed:

- the modeling of process-aware hyperlinks;
- the modeling of non-navigational sequences of views in process-flow mode and
- the modeling of associations between the pages in the navigation model and the related activities in the process model.

In Fig. 8 is depicted a model for the make review (sub) process that uses a simple extension of a WAE [10] navigation diagram (a stereotyped UML class diagram that is part of the user experience model) and a EPML diagram. We decided to use WAE for mainly two reasons: first, to remain agnostics with respect to other web applications development methods like [9, 15, 26, 27] that have been extended

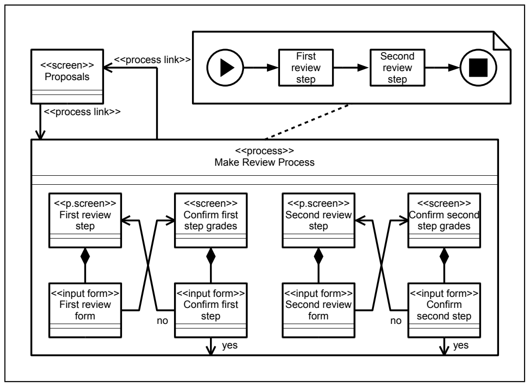

<<screen>> Proposals
<<process link>>
<<process link>>
First review step
Second review step
<<process>> Make Review Process
<<p.screen>> First review step
<<screen>> Confirm first step grades
<<p.screen>> Second review step
<<screen>> Confirm second step grades
<<input form>> First review form
<<input form>> Confirm first step
<<input form>> Second review form
<<input form>> Confirm second step
no
yes
no
yes

Fig. 8. Modeling the review sub-process

in order to model process-aware applications and, second, because it is a quite "lightweight" method that easily allows simple extensions. This is mostly because WAE is not targeted at model driven development and, thus, the design models are not overloaded with details.

From this example it is easy to see how the aforementioned issues can be addressed. Process-aware hyperlinks are modeled with navigational associations that connects to a *process*-stereotyped class. In order to remark the specific behavior of these associations we also used the *process link* stereotype, but it is not really essential. The *p.screen* stereotype is used (for inner classes inside a *process*-stereotyped class) to mark the entry points of sub-sequences related to an action in the EPML model. The active sub-sequence can be easily inferred since the name of the *p.screen*-stereotypes classes correspond to the names of the action in the process model. When a navigational subsequence in process-flow mode returns the control to the process-flow, a navigation connection is made between the last screen (or its aggregated forms) and the outer process class that contains it (as in the two input confirmation forms of the example). Process-stereotyped classes admit only one exit association that leads to the subsequent page (or process) that is visited when the (sub)process ends. In our example this association too is process link-stereotyped.

As far as implementation strategies are concerned, several options are available, depending on the used web application framework. The solution we present here is based on Java EE (which is a natural choice, since the EPML engine is written in Java) and the Struts MVC framework; similar solutions, however, can be implemented with different technologies. Basically, the functions that have to be supported by the framework in order to address the issues related to process-aware web applications that we mentioned above are:

- implement process-aware links to create, resume or join a process;

- dispatch the correct sequence of pages in process-flow mode;
- hide process-aware links that would lead to processes in which there are no actions (or there is not a specific action) assigned to the current user.

When using a MVC framework like Struts these issues can be easily addressed by writing specific process-aware controllers that, interfacing with the engine, dispatch the correct view given the current user (in our implementations we assume this information can be extracted from the user's session data) and the process the link points to. In the case of processes in which more than one action can be associated to the same user (this is an event that does not show up in our example) an additional parameter specifies which, among the available actions, has to be used as a starting point for the current flow. Process-aware hyperlinks can then be created simply pointing to a process-aware controller. The same controller can be used to dispatch the correct sequence of pages in the process flow. Finally, generic parts of a web page can be hidden by using a conditional custom tag that queries the engine about the availability of an action for the current user in the target process.

The management of the data perspective introduces some subtle issues and, while it is slightly out of the main topic of this article, we are going to discuss possible solutions. The main problem, in this context, is that almost every software system has some kind of data perspective; this is the case for EPML too, since some kind of data management is needed at least to allow the processors to take decisions when needed. It turns then out that we are trying to mix two sub-systems that have two distinct data perspectives. Duplicating the application data in both perspectives is unfeasible and error prone, so reasonable solutions are: using one of the two data perspectives and adapting the other system (if possible) to interface to the selected data management solution or using an external sub-system that addresses the data perspective and adapting the remaining sub-systems to interface to it. The correct solution depends on the specific overall architecture. In our case, for example, if we are dealing with a large business process that interfaces with several systems and is just partially participated using a web application, then using the existing data perspective and design the web application in order to interface to it is the better solution. On the other side, if we are dealing with a fully web-based business process, using the data perspective that is managed by the web application framework (in the case of Java Enterprise Edition, session data for the web tier - persistent data via EJBs or Hibernate for the business tier) is a feasible solution. Assuming the second scenario in our example application, Java Enterprise Edition-based solutions are used to address the data perspective: then, in this case the process logic of the EPML processors has to be designed so that it can access the web framework data (in other words the processors in EPML have to be able to interface to HTTPServletRequest-accessible data - query strings, form submitted-data, session data, etc. - and to the model components). Designing this solution is trivial using Java Enterprise Edition and the EPML engine in which we simply have to add the relevant references into the data bag of the token associated to the current flow each time a terminate activity event is dispatched to the engine (typically when a process-aware controller is activated as a consequence of a process-related interaction returning the control to the process flow). In the project proposal review application this happens, for example, when the reviewers confirm the first review step. A processor positioned between the *First review step* and the *Second review step* activities (which is actually present in our example but is not shown in the diagram for conciseness) takes decisions about outgoing flows by accessing the form data filled by the reviewer. In our prototype, to better separate the two sub-systems, the relevant web framework data are used to create an XML document that is put in the flow data. The processor uses XQuery as seen before for EPML.WS.

7. EPML.SIM

Business Process Simulation (BPS) is widely acknowledged as an effective technique to increase the chance for success of Business Process (re-)Engineering projects and, in general, to drive strategic business decisions.

In this context, we have designed and implemented EPML.SIM, a tool for modeling and simulating processes based on EPML. In particular, we propose to use EPML for modeling the control-flow perspective of a process, while ancillary (potentially pluggable) software systems can be used to support the remaining perspectives and to drive the discrete event simulation. This approach allowed us to design an effective simulation tool with a minimal footprint.

7.1. Architecture of the simulator

The simulation tool is built around three main components: the EPML engine, a *driver* with the responsibility of managing the (simulated) events (events generated by the environment, end task events, timeouts, ...) and a pluggable resources model. The driver also initializes the engine by specifying the process model to be enacted, its initial state and how the process logic and the code of the activities have to be overridden for simulation purposes. The tool, in fact, assumes that the EPML model is a generic model, possibly designed to support the enactment of the process in a real software system (and, in fact, a snapshot of the state of a running process can be used as the simulation's initial state).

The original process logic (that is the code associated to the processors) could reference data and components that are not available during the simulation. As an elementary example, consider a processor that implements a routing decision by analyzing the data produced by a previously run activity. In this case the processor's code can be overridden for simulation purposes with a pre-defined component that performs the decision on a random basis given a probability for each of the outgoing branches, thus obtaining the behavior of most of the aforementioned simulation tools. If a more detailed modeling is required, the processor's code can also be overridden by a code snippet (EPML directly supports all the languages compliant with JSR 223 which includes Java, BeanShell, Groovy, JavaScript, Python, Ruby, TCL, XPath and others), even a code accessing the cell of a spreadsheet in order to take its decision (we actually used this solution in a simulation where a detailed financial model of the organization was available).

As stated above, also the code associated to the activities can be overridden. Pre-defined activities returning only duration informations on the basis of a probability distribution (chosen among constant, uniform, normal, gamma and exponential) or by historical data stored in a text file or in a column of a spreadsheet, are available. If a more detailed model is required, any kind of code can be used (as seen before for the processors), allowing the interaction with other business models (or with other software systems). Using this technique we can address the functional perspective.

The data perspective is addressed by using the basic data handling capabilities provided by the EPML engine. In EPML tokens are used to keep track of the status of a process. The EPML engine allows tasks to access a *data bag* (implemented as an associative map) associated to the token(s) that activate the task. The data bag is also used to carry information about the items processed during the simulation. This is implemented by specifying which events are used to generate new items (for example the event representing the arrival of a new request to be processed can be tagged as a request generator). Tokens enabled by these special events are tagged (by adding the relevant information to their data bag) and are subject to statistical recording by the driver component.

The organizational perspective is implemented by a pluggable component (a Java class implementing a specific interface). The driver queries this component when a new activity has to be executed in order to obtain the needed resources (and to know their costs). The component is also notified when a resource is (or more resources are) released (because of the end of an activity). A basic component is provided which implements a simple role-resources matrix combined with an availability matrix. Yet again, if a more detailed organizational model is required a new Java class implementing the details of this specific model has to be created. Notice, however, that this solution does not support a resource model with preemption (that is a resource model in which a resource can be reclaimed by a high priority task while performing a lower priority one, which is canceled). This is because in EPML.SIM the canceling of a task is possible only from the control-flow perspective, that means that when such a behavior is required it has to be implemented in the process model (by using the cancellation support of EPML).

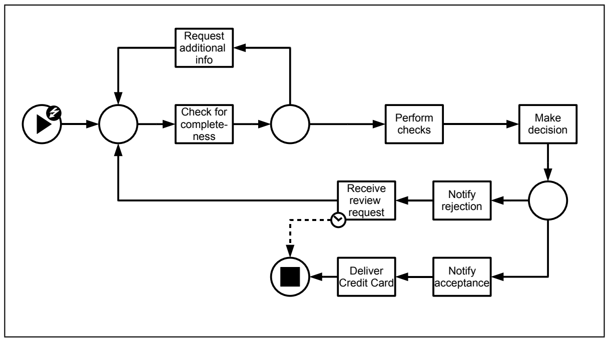


Fig. 9. The credit card application process

7.2. Simulating a Business Process with EPML.SIM

In this section we show how a Business Process can be effectively simulated by using EPML.SIM. To this end we use, as an example, the credit card application process depicted in Fig. 9 (inspired to the example from [29]).

The behavior of the modeled process is quite straightforward; the event associated to the start processor represent a new incoming application and is used to generate a new instance of the process; the clock icon associated to the `Receive review request` represents a time-out: if the activity is not performed within a specified amount of time the path exiting from the clock icon (an exception path) is activated.

This diagram corresponds to an XML file (that can be produced with EPML modeler, a graphical editor created with Adobe Flex). In this example we call this file process.xml. This XML file, along with another XML file (simulation.xml) specifying which (and how) processors/activities have to be overridden and, optionally, with an initial state description, are used by EPML.SIM. In our example we suppose that the activity `Check for completeness` is characterized by a duration depending on a normal probability distribution. We then override the behavior of this activity in simulation.xml with the following XML fragment:

```
<activity activityId="CheckForCompleteness"
```

```
    classname="epml.simulator.executors.
               ProbabilityActivityDuration">
  <param name="distribution">normal</param>
  <param name="unit">minute</param>
  <param name="mean">5</param>
  <param name="stdDev">2</param>
</activity>
```

where `ProbabilityActivityDuration` is the name of a pre-defined Java class that returns the duration of the activity by using a probability distribution (normal, in this case, with mean 5 and standard deviation 2).

For the subsequent processor (implementing a choice between two possible paths in the process structure) we can use a similar approach by redefining it in simulation.xml as follows:

```
<processor processorId="choice1"
    classname="epml.simulator.executors.
               ProbabilityProcessSwitch">
  <param name="e6">.1</param>
  <param name="e7">.9</param>
</processor>
```

where `ProbabilityProcessSwitch` is the name of a pre-defined Java class that implements a random choice between the outgoing edges, listed as parameters, associated to different probabilities (*e6* and *e7* are the ids of the outgoing edges as defined in process.xml).

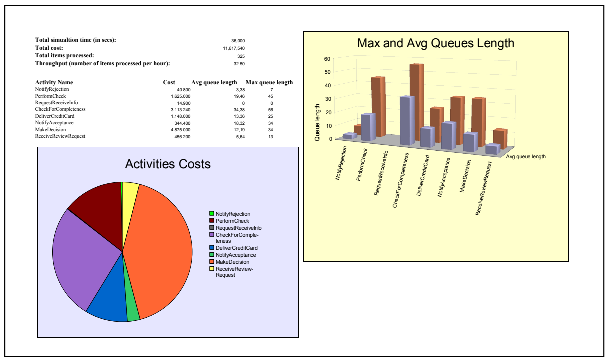
By using a similar approach for other activities and processors, and by setting up the details of the built-in resources model, it is possible to create a simple simulation. Please notice that, at the time of this writing, the file simulation.xml has to be edited by hand. An extension to the EPML modeler tool to set up the details of a simulation is in the works.

As stated above, however, EPML.SIM allows to produce more accurate simulations where more detailed models are available. As an example consider the `Make decision` activity and its subsequent processor. We could override the activity using the `ProbabilityActivityDuration` and the processor using the `ProbabilityProcessSwitch` as seen above. In this case, however, we suppose we want the decision being implemented by the action dependent on a parameter representing the request rate (calculated in another point of the process and added to the token's data bag). We can use two approaches: override the code associated to the processor with an ad hoc script fragment or override the code associated to the activity. In this second case, assuming that the processor implements the decision by analyzing the information returned by the `Make decision` action, we have to override the action so that it returns the same values assumed by the processor. This latter approach is best suited for when we use a process model created for the actual enactment of the process. An example is as follows:

```
<activity activityId="MakeDecision"
    classname="epml.simulator.executors.
    ActivityInstanceExecutor">
  <param name="language">BeanShell</param>
  <param name="code">
    <![CDATA[
    Double requestRate = ((Double)
      (((Map)tokenData.
      get("simulation")).
      get("requestRate"))).doubleValue();
    boolean accept = requestRate > 10;
    ((Map)tokenData.get("transient")).
      set("return", new Boolean(accept));
    ((Map)tokenData.get("simulation")).
      set("duration", new Double(10));
    ]]>
  </param>
</activity>
```

where *transient* is a special entry in the token's data bag that is used for information that can be discarded when a new activity is executed and is typically used to store the return values of an activity. The *simulation* entry is also a special entry that is used to carry information related to the simulation (in this case for accessing the previously calculated request rate and for setting the simulated duration, in seconds, of the activity). Please notice that the code in the example may look complex at first sight because of the way Java (and thus BeanShell) accesses maps; language verbosity aside it just sets and retrieves values from (nested) associative data structures.

Once the models have been set up, the simulation can be launched. To this end an horizon has to be decided. In EPML.SIM the analyst can decide to stop a simulation after a specified amount of (simulated) time, at a specific date/time, after having processed a certain amount of items or by using a generic script that is called after the processing of each event (when the script returns a false boolean value, the simulation ends). Support for ending a simulation when a service level agreement (related to the duration of the activities, to the utilization of the resources or to the costs) is not met is on the works. At the time of this writing EPML.SIM does not support animations to give visual feedback on the simulation's progress. At the end of the simulation a report is produced. The report can be either in plain text format or in Open Office Calc format (which includes charts and is formatted in such a way that it can be used to easily generate a PDF report). A fragment of a sample report is depicted in Fig. 10. In a report are shown the simulation parameters and results, such as the total simulation time, the total cost, the throughput, how many activities (or items) have been simulated and, for each activity, its cost and the maximum and the average queue length.



Total simualtion time (in secs):	36,000
Total cost:	11,617,540
Total items processed:	325
Throughput (number of items processed per hour):	32.50

Activity Name	Cost	Avg queue length	Max queue length
NotifyRejection	40.800	3,38	7
PerformCheck	1.625.000	19,46	45
RequestReceiveInfo	14.900	0	0
CheckForCompleteness	3.113.240	34,38	56
DeliverCreditCard	1.148.000	13,36	25
NotifyAcceptance	344.400	18,32	34
MakeDecision	4.875.000	12,19	34
ReceiveReviewRequest	456.200	5,64	13

Fig. 10. A report produced by EPML.SIM

8. RELATED WORK

In the last few years, a large spectrum of workflow languages and process modeling tools ([14, 6, 4, 19, 16], to name a few) have been proposed from the industry and the academia. Most of these solutions use different strategies to model a process and each of them shows specific strong and weak points, making hard to compare the different solutions. We propose to use expressive power and suitability as reasonable metrics to this end. As far as expressive power is concerned most of the proposals show strong limitations. For example, on the basis of our hands-on experience, common real-world processes turn out to be very hard (if at all possible) to model with most existing languages without modifying its semantics or producing an overmuch complex specification. Suitability too is related to expressive power but it is also related to the ability to adapt to different application domains. This is often related to the ability to be integrated in different software architectures. Some of the most recent proposals (like YAWL and Orc [14] - all coming from the academia) try to address the problem of expressive power by implementing all (or most of) the classic workflow control-flow patterns (the first language to claim to support all the patterns is YAWL, not surprisingly designed by the same research group that originally defined the patterns, what is surprising is that this claim is still not supported by a proof). YAWL is a powerful language and comes with a formal semantics (that is actually used to check specifications properties) and a reference implementation. Its main limits are related to its suitability outside the workflow domain, both because of its Petri nets-inspired model, and because of its engine architecture.

Another proposal that deserves to be referenced is BPMN. BPMN is a OMG-endorsed specification that is receiving a great deal of attention by the industry. This is mostly due to the fact that a large number of process based applications, and a large number of software products to develop them, already exists. Most of these applications are related to business process management (ERP, workflow, supply chain management) and they are urged to support a high degree on interoperability. In this context a process modeling-related standard is badly needed. The problem with BPMN is that it tries to address too many issues. It presents itself as a tool for high level - conceptual process modeling that can be used to outline a process without defining its detailed semantics but it also claims to support MDA-like translations into executable specifications (in this case using BPEL). But in the article we already pointed out the limits of this approach. It is our opinion that BPMN is a good modeling notation (the "UML for processes") but it falls short when it comes at programming in the small.

9. CONCLUSION AND FUTURE WORK

The wide array of applications, belonging to different domains, that we designed and implemented by using our EPML-based framework witness that a modular approach to process-aware application is possible from both a design and an implementation point of view. This implies that the large number of existing process modeling languages and systems cannot be justified only by the large spectrum of application domains. In our opinion this is mostly due to the lateness of the academia with respect to the needs of the industry: the latter, lacking strong indications from the former, went its own way proposing a large number of inherently limited tools. The separation of concerns, in the form of a clear distinction between computation, interaction model and process logic, that is at the roots of EPML, provided a solid framework for achieving a high level of modularity. We hope that these concepts can help in defining the priorities around which next generation process modeling languages should be designed (or around which current languages should evolve, as in the case of a possible forthcoming executable BPMN specification).

Our work on EPML continues. While the language has reached a good level of maturity and no major changes are foreseeable the runtime-system and the support tools are subject to a continuous evolutions. For example: while a graphical modeling tool based on Adobe Flex is already available, another tool based on Eclipse is on the works. As for the runtime-system: the engine supports state saving/restoration by using a relational database as a backend; checkpointing and recovery is still not fully implemented. These are just examples of a long todo list that never shrinks as new possible applications of our framework continue to emerge.

10. REFERENCES

[1] Activebpel open source engine project. `http://www.activebpel.org/`. Accessed January 2009.

[2] BPELJ: BPEL for Java technology. White paper. Available at `http://www.ibm.com/developerworks/library/specification/ws-bpelj/`. Accessed January 2009.

[3] Business Process Execution Language for Web Services version 1.1. `http://www.ibm.com/developerworks/library/specification/ws-bpel/`. Accessed January 2009.

[4] W. M. P. V. D. Aalst and A. H. M. T. Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[5] W. M. P. V. D. Aalst, A. H. M. T. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(14):5–51, 2003.

[6] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guizar, N. Kartha, C. K. Liu, R. Khalaf, D. Konig, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`. Accessed January 2009.

[7] F. Arbab. What do you mean, coordination? Technical report, Bulletin of the Dutch Association for Theoretical Computer Science, NVTI, 1998.

[8] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[9] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Trans. Softw. Eng. Methodol.*, 15(4):360–409, 2006.

[10] J. Conallen. *Building Web Applications with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Commun. ACM*, 35(9):75–90, 1992.

[12] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[13] S. Jablonski. Mobile: A modular workflow model and architecture. In *Proceedings of the 4th International Working Conference on Dynamic Modelling and Information Systems*, 1994.

[14] D. Kitchin, W. Cook, and J. Misra. A Language for Task Orchestration and its Semantic Properties. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, pages 477–491. Springer Berlin / Heidelberg, 2006.

[15] N. Koch, A. Kraus, C. Cachero, and S. Meliá. Integration of business processes in web application models. *Journal of Web Engineering*, 3(1):22–29, 2004.

[16] OMG. Business Process Modeling Notation (BPMN) version 1.0. `http://www.bpmn.org/`.

[17] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, and W. M. P. van der Aalst. From BPMN process models to BPEL Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2006)*, pages 285–292, Washington, DC, USA, 2006. IEEE Computer Society.

[18] G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:330–401, 1998.

[19] D. Rossi. X-Folders: documents on the move. *Concurr. Comput.: Pract. Exper.*, 18(4):409–425, 2006.

[20] D. Rossi and E. Turrini. EGO: an E-Games Orchestration Platform. In *Proceedings of the 8th annual European GAMEON®Conference on Simulation and AI in Computer Games*. EUROSIS-ETI, 2007.

[21] D. Rossi and E. Turrini. EPML: Executable Process Modeling Language. Technical Report UBLCS-2007-22, Department of Computer Science, University of Bologna, 2007.

[22] D. Rossi and E. Turrini. Using a process modeling language for the design and implementation of process-driven applications. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2007)*. IEEE Computer Society, 2007.

[23] D. Rossi and E. Turrini. Designing and architecting process-aware web applications with EPML. In *Proceedings of the ACM symposium on Applied computing (SAC 2008)*, pages 2409–2414, New York, NY, USA, 2008. ACM.

[24] D. Rossi and E. Turrini. An executable language/enactment engine approach for designing and architecting process-aware web applications. *International Journal of E-Business Research (IJEBR)*, 5(3):1–13, 2009.

[25] N. Russell, A. H. ter Hofstede, W. M. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org, 2006.

[26] H. A. Schmid and G. Rossi. Modeling and designing processes in e-commerce applications. *IEEE Internet Computing*, 8(1):19–27, 2004.

[27] O. D. Troyer and S. Casteleyn. Modeling complex processes for web applications using WSDM. In *Proceedings of the Third International Workshop on Web-Oriented Software Technologies*, 2003.

[28] W. M. P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 19–128. Springer, Berlin / Heidelberg, 2000.

[29] M. T. Wynn, M. Dumas, C. J. Fidge, A. H. M. ter Hofstede, and W. M. P. van der Aalst. Business process simulation for operational decision support. In *Proceedings of the Third International Workshop on Business Process Intelligence (BPI 2007)*, volume 4928 of *Lecture Notes in Computer Science*, pages 66–77, Berlin / Heidelberg, 2007. Springer-Verlag.