Abstract

Process driven software architectures 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 driven software architectures all of which, as the name suggests, are characterized as having a process perspective. In this paper we show that a (well designed) process modeling language and its execution engine can address the process perspective in different architectures and be (re)used among different application domains resulting in useful tools for the design and the implementation of process-driven applications. This can be achieved by maximizing the suitability of such a language via high expressive power and good separation of concerns. We also show sample process driven architectures, addressing different application domains, that are built with it.

1 Introduction

Process driven software architectures 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 architectures. The most studied among these architectures (mostly for historical reasons) are the WFMS. In the workflow community the idea of decomposing a system into perspectives is now well accepted [1]. The process perspective defines the coordination that has to take place between the actors and/or the software components participating in the workflow. The idea of decomposing a system into perspectives can be applied to different classes of process driven architectures as well, all of which (because of their own nature) will result in having a process perspective.

Just like a WFMS can be designed by integrating software components that support the various perspectives, the same approach can be applied to other process driven systems. We argue that a well designed process modeling language and its execution engine, addressing the process perspective, can be used across these different systems. In this paper we show how we designed these tools in order to maximize their suitability and we show sample process driven architectures, addressing different application domains, that are be built around them.

This paper is structured as follows: section 2 outlines the main issues we tried to address in designing our proposal; section 3 introduces YAWN, a novel PML we have designed; the sections 4 and 5 show two sample architectures using YAWN; section 6 discusses related works. Section 7 concludes the paper.

2 Coordination for process-based applications

Before presenting our proposal it is worth noticing that existing workflow systems, even when they are intended to support similar kind of processes, all use different mechanisms for their definition using specific workflow languages, which are (potentially graphic) formalisms to model the process that the WFMS has to enact.

Most existing workflow languages are exogenous [17] coordination languages: they define the interactions that can take place among actors (but they cannot change them at run time like, for example, Manifold [18] and other control-driven [3] coordination languages can do). From our point of view the workflow languages can be seen as instances of a more generic class of coordination languages we call process modeling languages (PMLs), that define the structure of the process the architecture has to support (other instances include web services orchestration languages). Pro-
cesses modeled with a PML can then be enacted by specific software components, usually referred to as engines.

While PMLs could potentially be instances of different coordination models most existing proposals are flow languages (in the broad sense of languages that describe the process in terms of, potentially concurrent, flows of execution, their interactions and their synchronizations). We restrict our interest to this category. Of course there is not a “one fits all” coordination model; flow languages, however, exhibit a high degree of suitability in this context.

In this paper we show how a well designed process modeling language and its execution engine can support a large spectrum of process based applications. The language by allowing the modeling of complex processes with a relatively simple graphical notation, the engine by defining a clear interface with other components in order to ease its integration in complex software architectures. In order to achieve this result we designed the YAWN language and its execution engine guided by the following criteria: high expressive power (or expressiveness, in the sense of [4]); clear separation of concerns; well defined interface model between the engine and its surrounding architecture. The first two criteria are analyzed in the following subsections, the third is discussed in subsection 3.1.

2.1 Expressive Power

The expressive power of workflow languages has been subject to several investigations in the last few years. The main problem with this kind of languages is that there is not a formal metric to evaluate it. One of the most successful approaches is the one based on the workflow patterns [4]; this is an analysis strategy that evaluates the languages with respect to their ability to model a set of predefined (sub)processes. The expressive power of a PML is a critical parameter to increase its suitability. A language that cannot easily model the interactions within a process surely poses large limitations to its usage. Most existing workflow languages, for example, cannot easily model a large number of common real-world interactions. In the design of YAWN we addressed the workflow patterns but we also striven to achieve a good compromise between simplicity and power.

2.2 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 [2]. In a coordinated process the computation is carried out by the software components or the human beings (actors) that participate in the process; while the coordination is carried out by the engine. We soon realized, however, that this distinction is too coarse. In our vision, 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. Despite of the importance of separating coordination and computation, many PML do not address this issue, since often the process logic is delegated to the activities.

YAWL [9] and BPEL are notable instances (coming from the academia and the industry, respectively) of PML (the former targeted at workflows, the latter at web services orchestration) that show this problem. Both YAWL and BPEL use XPath predicates for basic control flow decisions but as soon as the process needs to take decisions that cannot be expressed via a simple XPath expression, the decisions are delegated to activities that have to be created ad hoc for this task (from a technological point of view this is troublesome since the programmer has to create a web service to assist the process logic). The result is that the control flow logic of a workflow is spread between the workflow language and some external component. Implicitly acknowledging this limitation all existing commercial BPEL engine implementations we are aware of include custom extensions that allow to activate software component implementing the process logic. For the same reason, BEA and IBM (the same partners that supported the development of BPEL) proposed an extension of BPEL: BPEL-J [16]. In BPEL-J, Java code snippets 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, workflow engines must be extended in order to be able to activate Java computations (which may not be trivial if they are not written in Java themselves).

In the case of YAWN the interaction model is expressed with a (graphical) specification, the process logic can be implemented in any language provided that specific adapters are available to interface to the engine.

3 YAWN

YAWN is a graphical PML 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.

The purpose of this section is not to provide a detailed description of all YAWN features but only to give flavor of it. Interested readers can refer to [20].

The components of YAWN are shown in Fig 1.
A process specification modeled with YAWN can be enacted by means of an XML document. It is possible to translate a graph’s diagram in its XML representation. A process specification (in form of XML document) can be executed by an engine, which is a software component called YAWN engine; it takes as input an XML representation of the YAWN specification and it executes it. The YAWN engine is a software component written in Java 1.5 (and thus portable to most platforms). It has been designed as an event-based architec-

Two types of nodes exist in YAWN: activities (represented with squares) and processors (represented with circles). Activities are elements of computation: they can be either external applications or work items allocated to a workflow participant (possibly a human actor). The activities are triggered by the processors. Processors are elements of coordination: they implement the process logic using a standard programming language. The process logic specifies the routing of the process flow(s). Both activities and processors can produce data that will be employed by the following nodes.

A process specification modeled with YAWN can be a diagram or an XML document. It is possible to translate a graph’s diagram in its XML representation. 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.

A processor can be decorated with the decorations shown in Fig. 1; decorations allow the implementation of control-flow operations like choice, split and join. Each time a processor node is reached by an execution flow (and its associated join, if present, has been satisfied) it activates. Concurrent activations of the same processor are possible inside the same process. In the following paragraphs we briefly describe the semantics of every decoration that can be associated to processor nodes.

Six types of processors exist: simple processors, par processors, join processors, and processors, start processors and end processors.

The start processor and the end processor (obtained decorating a processor with a start and end decoration, respectively) represent the entry point and the exit point of a graph. The incoming-flow processor (obtained decorating a processor with a incoming-flow decoration) represents a point in a diagram where an external flow can enter.

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 over which the execution flow(s) will be routed.

A processor with a par decoration (par processor) enables to split a single process execution flow in parallel flows, and route them over 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 along the edge(s). If such a notation is not present, no limitation about the number of flows is imposed.

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 YAWN is quite sophisticated. In a YAWN 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). A join processor present in the par set reach the join processor. The par sets are calculated by means of an algorithm based on a network colouring mechanism (for sake of conciseness the algorithm is not described in this paper).

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 written inside the decoration) indicates the number of process flows the join processor has to wait before activating.

A processor decorated with an and decoration (and processor) implements a different kind of synchronization. An and processor does not maintain a par set; it activates as soon as a process flow, coming from every entry edge, reaches it.

3.1 The YAWN engine

A process modeled with YAWN can be enacted by means of a software component called YAWN engine; it takes as input an XML representation of the YAWN specification and it executes it. The YAWN engine is a software component written in Java 1.5 (and thus portable to most platforms). It has been designed as an event-based architec-
ture: it consumes events and produces both events and state transitions. An event can represent 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, incoming-flow events, and end-activity events. The start event triggers the execution of a new process instance; the incoming-flow event activates the incoming-flow processor within an existing process instance, 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 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 with 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 with 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 manipulates both the process instance data and the data produced by the previous nodes. 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 and can be integrated in YAWN in a way transparent to the engine.

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. The engine makes available to process logic mechanisms for manipulating process instance data and data produced by the previous nodes; at the same time the process logic informs the engine about which exit edge(s) has(have) to be activated. In the following example:

```xml
<processor name='checkCard' id='p7'>
  <logic language='java'>
      <![CDATA[
        String result=getResponse().getAttribute("result");
        if(result.equals("success")) {
            activateLink(getLinkByName("success"));
        } else {
            activateLink(getLinkByName("failure"));
        }
    ]]>  
  </logic>
</processor>
```

Two bindings for the process logic of the “checkCard” processor are shown. In the Java binding predefined methods (i.e. activateLink, getLinkByName) are invoked. The XPath binding assumes that data produced by the previous activity are in XML format or have been transformed into it.

## 4 YAWN.WS

YAWN.WS is a software architecture for web services orchestration based on YAWN. In order to support this specific applications class YAWN.WS complements YAWN with functions that address the interaction with synchronous and asynchronous web services. YAWN.WS poses itself as an alternative to system based on WS-BPEL. With respect to these systems YAWN.WS has the following advantages:

- graphical modeling;
- higher expressive power;
- ability to implement complex process logics without relying on external components.
In more detail: WS-BPEL has not a “native” graphical notation but there are specific guidelines about the usage of BPMN to model BPEL processes. It should be noted, however, that only a restricted subset of BPMN can be used to this end (which is obvious, given the fact that BPMN in more expressive than BPEL), the point is that which the subset is, is not clear at all (it is not stated in the BPMN specifications where the mapping to BPEL is addressed). The interested reader can found details about the BPEL-BPMN mapping (and how to overcome some of the problems it poses) in [15].

YAWN.WS is designed to be a component inside a J2EE architecture supporting JSR 109 and JSR 181. The application server takes care of associating incoming web services invocations (SOAP messages) to the engine’s input events, allowing the invocation of the YAWN managed process as a web service, and supporting the asynchronous replies in the management of the components participating in the process.

Process logic can, as usual with YAWN, be written in several languages. A specific binding for XQuery/XPath, however, has been implemented. With this binding an XQuery expression can be executed upon processor activation. Exit links can be activated by evaluating associated XPath expressions after the XQuery query has been run. Before the XQuery evaluation, an XML DOM document is built by aggregating process instance data and incoming reply SOAP messages sent by previously invoked web services. The XQuery expression is run on this data source; the result of this query is fed to the XPath expressions associated to exiting links. If no XQuery is present the previously described XML document is feed directly to the XPath expressions. The process designer can decide to use this specific binding or the usual ones (or to mix them).

To test YAWN.WS against BPEL we set up a simple experiment. We used a simple BPEL process (the load approval sample from ActiveBPEL [11] distribution) we “unplugged” the BPEL engine and we replaced it with YAWN.WS with success. Another interesting part of this experiment was the analysis the effort needed to setup a BPEL-based solution and the YAWN.WS one. We did not run formal tests to access a vague parameter like “effort” our experience with computer science students shows that the same task can be accomplished using YAWN.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 YAWN.WS).

YAWN.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 YAWN.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) is a software platform to deliver e-learning games based on YAWN. 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.

As stated above EGO is a web-based platform and it has been designed within the J2EE framework. Its structure is quite simple: the engine interacts with a web application hosted in a J2EE 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 submits its move, the move is processed by the Process servlet that produces a termination-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. 2 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 by activating the setup game processor; other players join via the new participant processor. Each new player is then assigned to the wait players interface, a simple passive web page that tells the user to wait for other players to join. This web page automatically reloads each few seconds so that when the assigned interface for this actor has changed the server has a chance to send the new web page. Once the and-par processor decides that the requested amount of players have joined, the actual game begins. Being a turn-based game
Figure 2. The beginning of a simple turn-based game

one player is in charge of making their move while other will have to wait. Once all players made their moves the game can continue to the next step.

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 Related Work

In the last few years, a large spectrum of workflow languages and process modeling tools ([7, 8, 9, 10, 12, 13, 14], 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 it 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, SMAWL [13] and Orc [7] - 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. SMAWL and Orc are interesting proof-of-concept languages but they provide only prototype implementations and tools.

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). This latter task needs a clear, unambiguous semantics, but this is not available (see for example [19]). 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.

7 Conclusions, limitations and future work

Several aspects of YAWN still have to be addressed. We are working to shorten the list. One of the main issues is related to the lack of a formal semantics. As stated above, a formal semantics is important for mainly two reasons: the ability to define the behavior of the system in a unambiguous way and the ability to check properties (such as reachability and liveness) of a specification. With respect to other proposals lacking a formal semantics (like BPEL and BPMN) YAWN has the advantage of having an existing reference implementation based on an event/transition semantics. We hope that this could lead to a relatively effortless translation into a transition system model.

Another inherent limit of YAWN, related to its model, is its inability to dynamically adapt the enacted process. This is usually a field in which rule-based models show an advantage (but a process subject to dynamic changes loses the correspondence between the running instance and its initial specification). Possible solutions include the transformation of a YAWN specification in a set of rules for a rule-based system or the adoption of dynamic modification of the network topology like in control-driven models. The latter is a fascinating opportunity we would like to investigate in the future.
Although some work still had to be done, YAWN can already prove that a PML and its engine, if well designed, can be used in a wide range of process based applications. In this paper we showed that with two sample architectures addressing web service orchestration and multi-player gaming (a more generic and a more specific application domain). 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. While arguably the “silver bullet” does not exist in this context (just like it does not in many field related to coordination models and languages) we hope our contribution can help in better understanding the issues that have to be addressed when proposing yet another workflow notation (by the way, this is what YAWN stands for).

References


