Patterns-based Evaluation of Open Source BPM Systems: The Cases of jBPM, OpenWFE, and Enhydra Shark

Petia Wohed[1], Birger Andersson[1], Arthur H.M. ter Hofstede[2], Nick Russell[3], and Wil M.P. van der Aalst[2,3]

[1] Stockholm University/The Royal Institute of Technology
Forum 100, SE-164 40 Kista, Sweden
{petia,ba}@dsv.su.se
[2] Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
a.terhofstede@qut.edu.au
[3] Eindhoven University of Technology
PO Box 513, NL-5600 MB Eindhoven, The Netherlands
{N.C.Russell,w.m.p.v.d.aalst}@tue.nl

Abstract. The abundance of approaches towards business process specification and enactment is well-known and is an ongoing source of confusion. One of the objectives of the Workflow Patterns Initiative is to provide insights into comparative strenghts and weaknesses of the state-of-the-art in Process Aware Information Systems (PAISs). Over the past years many approaches to business process specification including commercial offerings, modelling languages, and academic prototypes have been evaluated in terms of the patterns in order to assess their capabilities in terms of expressing control-flow dependencies, data manipulation and resource allocation directives. With the increasing maturity and popularity of open source software it seems opportune to take a closer look at such offerings in the Business Process Management (BPM) area. This report provides a patterns-based evaluation of three well-known open source workflow management systems: jBPM, OpenWFE, and Enhydra Shark.
Keywords: jBPM, OpenWFE, Enhydra Shark, Workflow Management Systems (WFMS), Business Process Management (BPM), open source software, workflow patterns.

1 Introduction

It is well-known and well-documented that there are a substantial number of approaches to process specification, originating both from academia and from industry, with differences in terminology, syntax and semantics. In the field of Business Process Management this situation substantially complicates the uptake of process technology as to determine which system is applicable in a specific context is not at all straightforward.

The Workflow Patterns Initiative [9], which started in 1999, has documented more than 100 patterns abstracting pieces of functionality supported by various workflow systems and process modelling languages. Patterns-based evaluations of a substantial number of approaches to process specification have been conducted which provide a comparative insight into relative strenghts and weaknesses. The patterns cover a number of so-called perspectives relevant to workflow modelling and enactment (see e.g. [10]) and as such cover the specification of control-flow dependencies [25], data definition and interaction [26], and various aspects of resource management [24].

A patterns-based evaluation of an approach to process specification determines to what extent there is *direct* support for the various patterns. Such evaluations are guided by explicitly provided evaluation criteria and, roughly speaking, they tease out the alignment between the concepts provided in the process specification approach and the various patterns and as such provide an indication of modelling effort required. Therefore, they are not concerned with expressive power, but rather with *suitability* (see e.g. [18]). Naturally, suitability is the more interesting notion as virtually any programming language is Turing complete and would allow one to specify arbitrarily complex workflows (see [1]).

With the growing maturity and popularity of open source software in general it seems opportune to investigate the capabilities of open source workflow systems. Whilst there are a large number of such systems available (for a couple of enumerations see [11, 19], each containing more than 30 offerings) relatively few seem to be of such a degree of maturity that they are used to support real-life processes. Three of the most widely utilised [8] open-source workflow management systems are jBPM [14], OpenWFE [21] and Enhydra Shark [7]. All of these systems are regularly updated and their web sites provide an indication of an active and sizeable user community. jBPM is part of JBoss (a commercial company), Enhydra Shark supports XPDL (the standard proposed by the WfMC) and OpenWFE is an active project on Sourceforge, labelled as "Production/Stable", and having more than 100,000 downloads[4]. In this paper we aim to provide insight into the state-of-the-art in workflow systems in the field of open source software

[4] Given (some of) the authors' close involvement with the YAWL Initiative, we refrain from commenting about the open-source workflow management system YAWL [2] in this paper.

through a patterns-based analysis of jBPM, OpenWFE and Enhydra Shark. For this evaluation the workflow patterns framework [9] is utilized: more precisely, the control-flow, the data and the resource patterns are used.

The paper is organized as follows. The next section briefly introduces the offerings under consideration: jBPM, OpenWFE, and Enhydra Shark. Sections 3 to 5 present the results for the control-flow, data and resource patterns evaluations, respectively. Finally, Section 6 summarises the results and concludes the paper.

2 An Overview of jBPM, OpenWFE, and Enhydra Shark

In this section an overview is provided of JBoss jBPM (Section 2.1), OpenWFE (Section 2.2), and Enhydra Shark (Section 2.3). The goal is to first characterize the systems, before evaluating them using the patterns.

2.1 JBoss jBPM

JBoss jBPM [14] is a workflow management system of which release version 3.1.4 was considered for the purposes of the patterns-based evaluation. The architecture of the system is sketched in Figure 1. It is based on the WfMC's reference model [5][5]. The system contains the following main components:

- A workflow engine called JBoss JBPM core component (also referred to as core process engine) which takes care of the execution of process instances.
- A process definition tool called JBoss jBPM Graphical Process Designer (GPD). It is a plugin to Eclipse, which provides support for defining processes in jPDL both in a graphical format and in XML format. jPDL (jBPM Process Definition Language) is the process language utilized by the system[6].
- JBoss jBPM console web application which has two functions. It is a web based workflow client whereby, in Home mode, users can initiate and execute processes (see Figure 3). It is also an administration and monitoring tool, which offers a Monitoring mode where users can observe and intervene in executing process instances.
- JBoss jBPM identity component (which as stated in Chapter 11.11 in [12], will be developed in the future), which will take care of the definition of organisational information, such as users, groups and roles to which different tasks can be assigned. Currently the definition of all this information is done through standard SQL insert statements directly in the workflow database.

A process definition in jPDL consists of a number of *nodes* and *transitions* between these nodes (for the graphical symbols in the language, see Figure 2). In addition, the concept of *Swimlanes* [12], borrowed from UML, is used as a mechanism for assigning tasks to users. As mentioned earlier, users are defined to the workflow database through SQL insert statements. The most important nodes in jPDL are 1) the *Start* and *End* nodes providing explicit representations of the beginning and the completion of a process, 2) the *State* node for modelling wait states, 3) the *Task Node* for capturing work packages, and 4) the routing nodes such as *Fork*, *Join* and *Decision* for performing parallel execution, synchronisation, and choice. A Task Node consists of one or more *Task* definitions. Tasks are atomic pieces of work either conducted by a human being or an application. Graphically only Task Nodes are shown and not the Tasks that they consist of. Furthermore, two constructs for hierarchical process decomposition are offered, namely *Super State* and *Process State*. A Super State is used to organize a group of related nodes into a unit. A Process State is used for indicating process invocation of subprocesses which are defined independently from and outside of the process they are invoked from.

There is also a special node type called *Node* through which facilities are provided for the introduction of custom code to define new desired behaviour. This construct can be used both for invoking external applications, as well as for adding new control-flow behaviour to a model. Important is that a Node is responsible for the further propagation of task instances passing through it.

[5] Interface 4, i.e. the interface towards other Workflow Engines, is excluded from this figure as it falls outside the scope of this evaluation.

[6] Processes can also be defined as Java objects or as records in the jBPM database (as stated in [12], Chapter 7 Persistence). However, these alternative notations, are outside the scope of our evaluation.

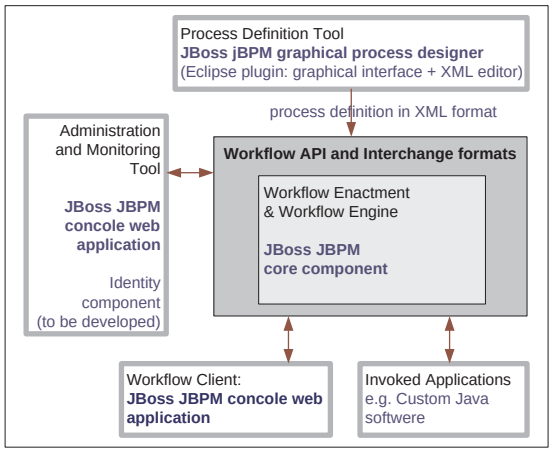


Figure 1 Architecture of jBPM

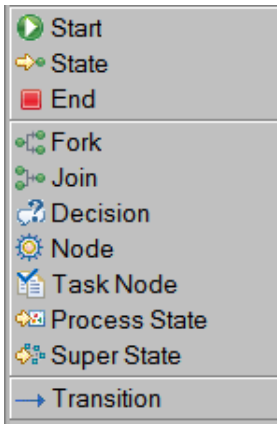


Figure 2 Symbols in jPDL

An alternative way to introduce customized behaviour is through *Actions*. Actions are defined through Java classes and specify customized behaviour, which is hidden in transitions or nodes and meant to implement behaviour important from a technical, but not from business perspective (in the specification, an update of a database is mentioned as an example of an action). In contrast to Nodes, Actions can not influence the specified control-flow and hence they are not responsible for any token propagation in a model. They are executed when the transition they are attached to is taken, or the node they are defined for reaches the state specified as trigger (which is one of the following: node-enter, node-leave, before-signal and after-signal). As Actions and Nodes require coding, they are not considered in this evaluation. (Recall that our evaluation is focusing on suitability, i.e., the question is not whether something is possible but whether it is directly supported without additional programming, etc.)

Listing 1 shows an example of a jPDL specification represented in XML and Figure 4 shows its graphical representation. In this example there is one swimlane 'ernie' which contains a user 'ernie' who is responsible for the whole process where first tasks A and B of task1-node execute in parallel and when both are completed task C of task2-node is started. Recall that the task nodes are graphically shown without their tasks, which in this particular example hides the parallelism present between tasks A and B in the flow. In the remainder of this paper closing tags in XML examples will be omitted when that improves readability. In addition, definitions of Swimlanes, Start state, and End state (i.e. lines 1-10 from Listing 1) will also be omitted so that examples can focus on those lines of code relevant to understanding pattern solutions.

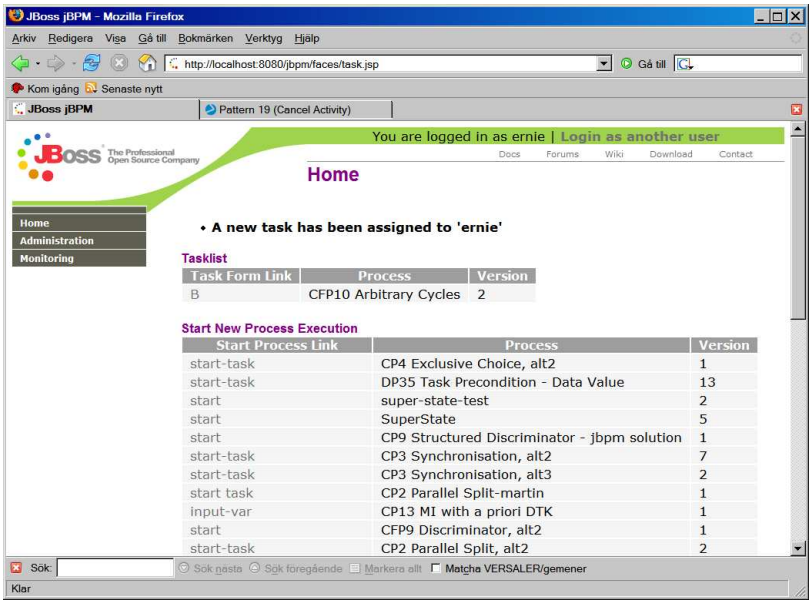
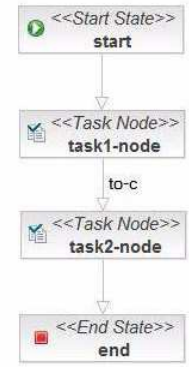


Figure 3 JBoss JBPM - Home window

Listing 1

```
<process-definition
  xmlns="urn:jbpm.org:jpdl-3.1" name="Synchronisation">
    <swimlane name="ernie">
       <assignment expression="user(ernie)"></assignment>
    </swimlane>
    <start-state name="start">
       <task name="start-task" swimlane="ernie"></task>
       <transition name="" to="task1-node"></transition>
    </start-state>
    <end-state name="end1"></end-state>
    <task-node name="task1-node">
       <task name="A" swimlane="ernie"></task>
       <task name="B" swimlane="ernie"></task>
       <transition name="" to="task2-node"></transition>
    </task-node>
    <task-node name="task2-node">
       <task name="C" swimlane="ernie"></task>
       <transition name="" to="end1"></transition>
    </task-node>
</process-definition>
```

Figure 4 JBoss JBPM - graphical representation of the model in Listing 1



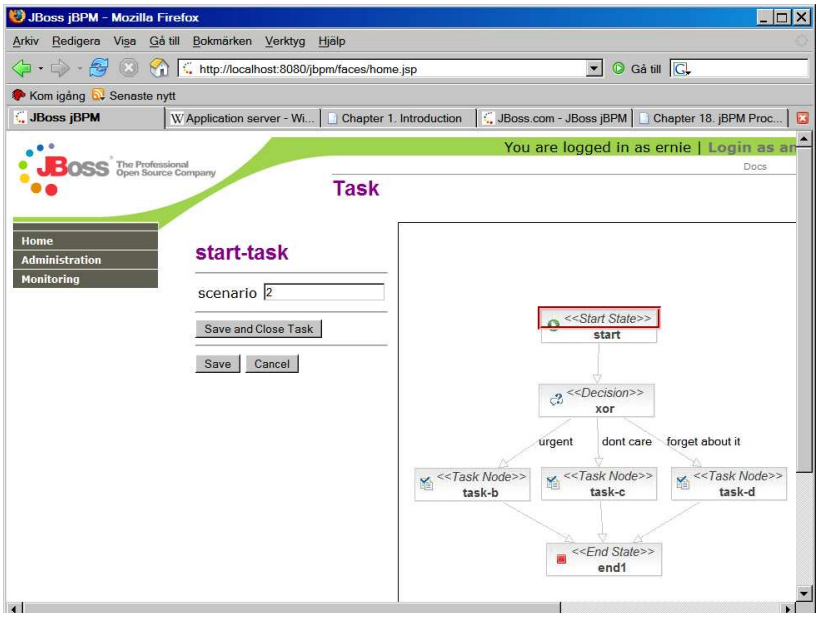


Figure 5 JBoss JBPM - Task window

The execution of a task instance is illustrated in Figure 5. When a task instance is being executed, the name of the task is shown as well as fields capturing its input and output data. In the example shown, there is an input field 'scenario' where the value '2' has been entered (the definitions of fields in jPDL will be explained in Section 4). In addition the corresponding task highlighted in a graphical representation of the workflow is shown, thus providing some notion of context. A task instance can be completed by choosing the 'Save and Close Task' option, it can be cancelled by choosing the 'Cancel' option in which case any data provided is lost, or it can be suspended by choosing the 'Save' option in which the data provided thus far is retained and work on the instance can resume at some later stage.

For this evaluation, version 3.1.4 of the tool was analysed (i.e. jbpm-starters-kit-3.1.4 package from 25th of January 2007). During late June a transition to an update of the tool was considered (i.e. jbpm-jpdl-suite-3.2.1 package from 25th of June 2007). However, the state of the Workflow Client following with this installation was considered as immature. For instance every user was able to execute every task irrespective of the actual process definition assigning specific users to the various tasks. For this reason we decided to stay with the earlier but more stable version 3.1.4 of the tool.

2.2 OpenWFE

OpenWFE [21] is a workflow management system, written in Java, of which release version 1.7.3 was considered for the purposes of the patterns-based evaluation. This system has the following main components (a sketch of its architecture, based on the WfMC reference model [5], is shown in Figure 6):

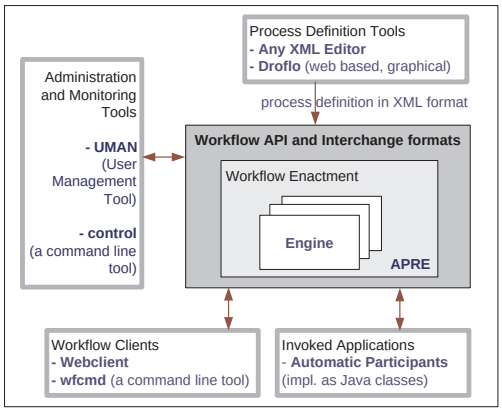


Figure 6 Architecture of OpenWFE

– An Engine routing work items to users in accordance with a process definition. Users are either human participants or automated agents. Automated agents are implemented in a language like Java or Phyton and defined as users in a workflow through a corresponding APRE, i.e. Automatic Participant Runtime Environment (Java classes are defined through Java APRE and Phyton script is defined through Pyya APRE).

– A web-based workflow design environment called Droflo which can be used to graphically define process models, which are then translated to OpenWFE's process definition language in XML format. OpenWFE uses its own process definition language. Due to lack of documentation of the graphical notation, in this evaluation the XML format was used exclusively.

– A Webclient (also called webappserver) providing the user-interface to the Engine and to the worklists to which the engine distributes the work.

– A web-interface for resource management, called UMAN (for User MANagment). Due to lack of documentation on UMAN, all resource management during our work was performed through direct modification of the files storing user information.

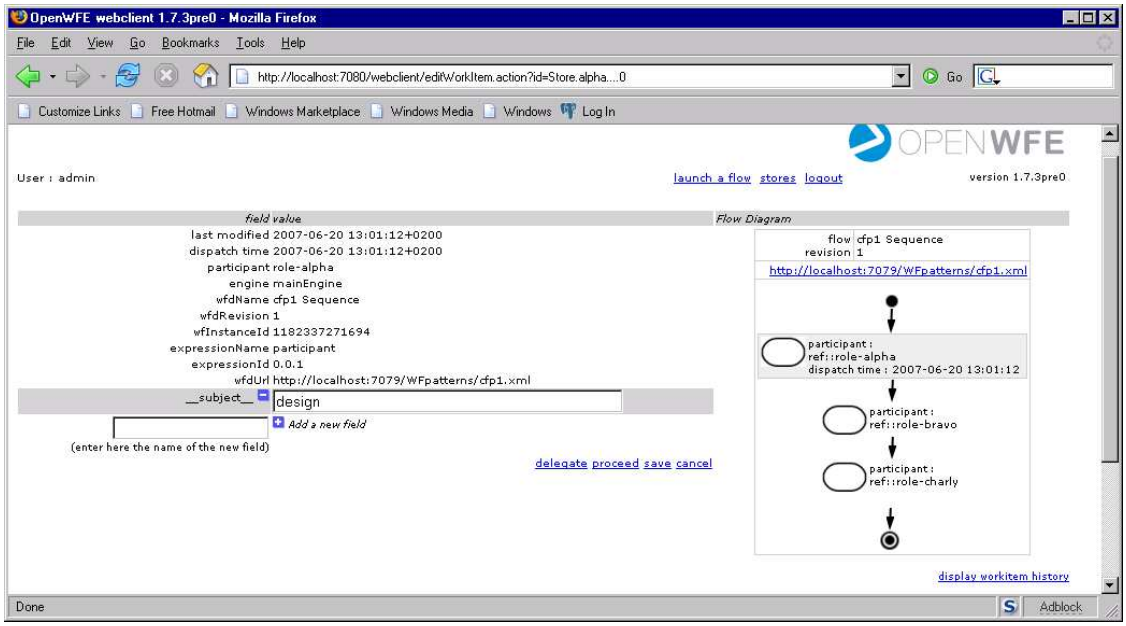


Figure 7 OpenWFE: executing a task

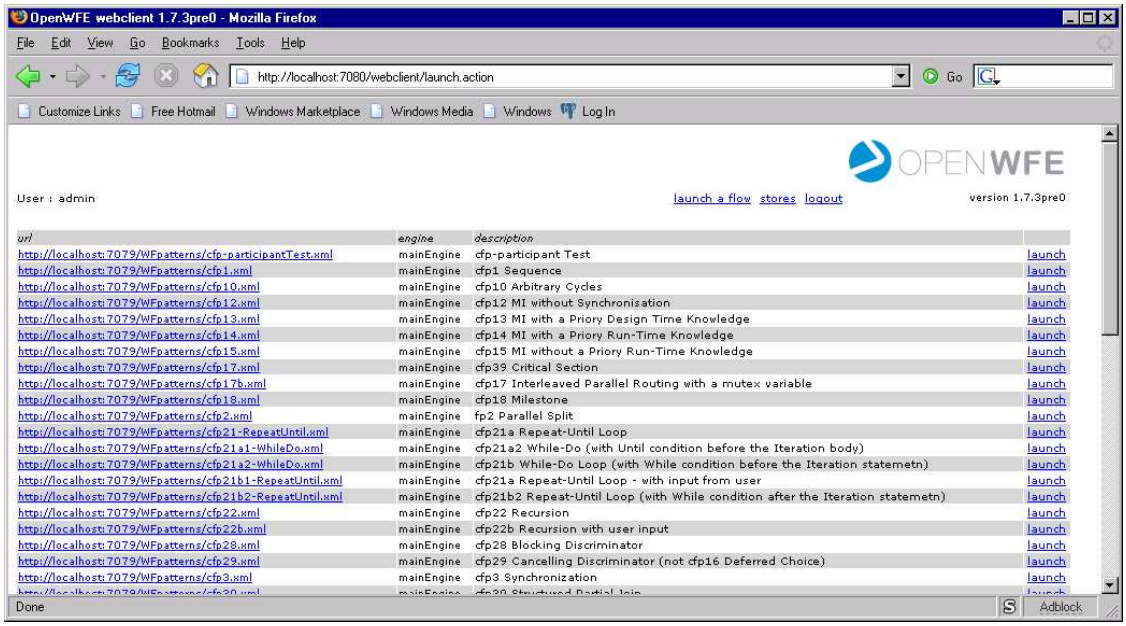


Figure 8 OpenWFE: launching a process

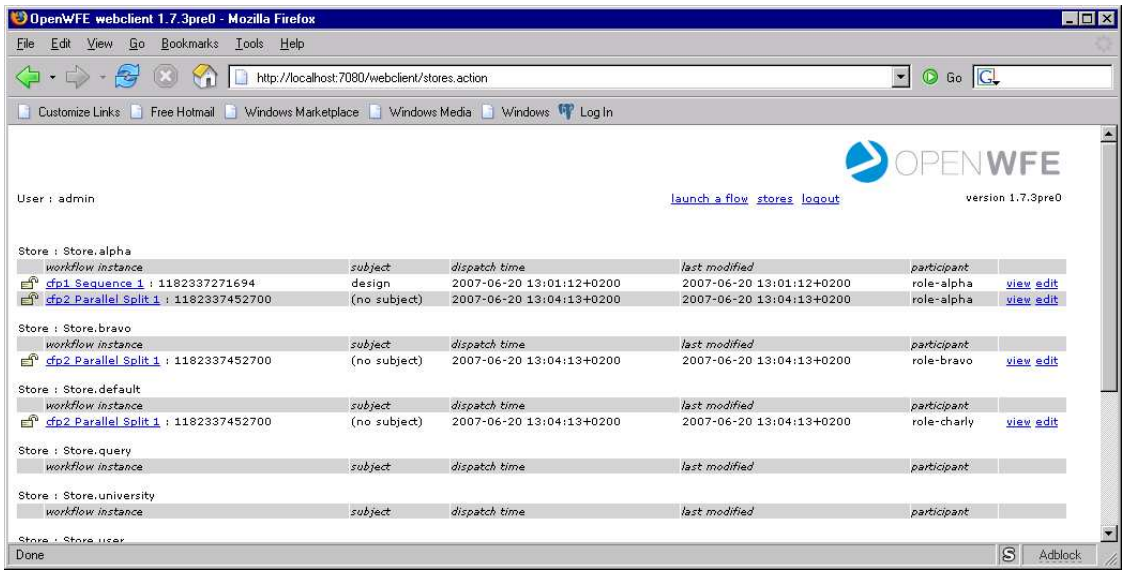


Figure 9 OpenWFE: accessing a work list

The development of OpenWFE has now migrated from Java to Ruby with the new implementation referred to as OpenWFEru. However, during our test of OpenWFEru, at the end of June 2007, a client for OpenWFEru similar to this for OpenWFE was not yet available. As the lack of such a client would considerably slow down our work, and as such a client is considered as an integral part for a WfMS, we decided to continue the work with the older Java version of the tool, i.e. OpenWFE 1.7.3.

Through the Webclient users access their worklist. The client allows them to view work items offered to them (and possibly to other users), choose a work item in order to execute it, and to report the completion of a work item. During process execution users can see a graphical representation of the process with the task they are working on highlighted (see Figure 7).

When logged in, users, also referred to as principals, can switch between the Launch view (see Figure 8), where they can launch processes for which they have the proper authorisation and the Stores view (see Figure 9), where they can access their work list. Users can have certain privileges (read, write, delegate) for stores, which themselves contain a number of participants. At design time work is assigned to participants which means that at runtime this work can be performed by any user that has write-access to the store of which this participant is a member. E.g. a work item

<participant ref=”role-alpha”> can be executed by user Alice if Alice has write-access to a store of which role-alpha is a member. The engine thus routes work according to the process logic, user privileges and participant membership of stores. In Figure 9 Store.alpha is an example of a store and role-alpha an example of a member participant. Listing 2 shows an XML excerpt illustrating how this membership relation is defined. Listing 3 shows the definition of user Alice with the privileges given to her to read, write and delegate work items from Store.alpha.

The main routing constructs used in OpenWFE are *sequence* for sequential execution, *concurrence* for parallel execution, various dedicated constructs for *loop*s (e.g. while, until, *iterator*), and the *concurrent iterator* to deal with multiple concurrently executing instances of a task. Attributes for these constructs allow further refinement of their behaviour. The language is block structured which imposes some restrictions on the specification of arbitrary forms of synchronisation and iteration (for a general discussion on this issue see e.g. [4]).

Listing 2 Fragment of worklist-configuration.xml

```
<application-configuration
   name="openwfe-worklist">
 <service
   name="Store.alpha"
   class="[..].SimpleWorkItemStore">
 <param>
    <param-name>participants
    </param-name>
    <param-value>role-alpha
    </param-value>
 </param>
 ...
 </service>
```

Listing 3 Fragment from passwd.xml

```
<passwd> ...
 <principal name="alice"
      class="[..].BasicPrincipal"
      password="+99-124+86+60">
   <grant name="store.alpha"/>
 </principal>
...
 <grant name="store.alpha"
      codebase="file:./jars/r/*">
   <permission
    name="Store.alpha"
    class="[..]StorePermission"
    rights="read, write, delegate"/>
 </grant>
...
```

Listing 4 A process definition

```
<process-definition
     name="cfp1 Sequence"
     revision="1.0">
 <description>1 Sequence</description>
 <sequence>
   <participant ref="role-alpha"
      description="design"/>
   <participant ref="role-bravo"
      description="implement"/>
   <participant ref="role-charly"
      description="deploy"/>
 </sequence>
</process-definition>
```

Listing 5 The process from Listing 4, alt 2

```
<process-definition
    name="cfp1 Sequence"
    revision="1.1">
  <description>cfp1 Sequence</description>
    <sequence>
       <set field="__subject__" value="design"/>
       <participant ref="role-alpha"/>
       <set field="__subject__" value="implement"/>
       <participant ref="role-bravo" />
       <set field="__subject__" value="deploy"/>
       <participant ref="role-charly" />
    </sequence>
</process-definition>
```

A process can be decomposed into *subprocesses*. A subprocess can be defined as part of a main process or it can be specified independently. In the latter case it can be invoked by multiple processes.

In Listing 4 the specification of a simple process containing a sequence of activities is shown. In a process specification work items are assigned to participants (<participant ref=“role-alpha”/>) and routing constructs, such as <sequence> in the example, are used to control the order in which these items can be performed. The description attribute (e.g. description=“design”) is optional and it aids in understanding what the task is about. Unfortunately, the description cannot be seen by users viewing their work list (Figure 9), it can only be seen when a work item is started. This makes it hard for users to understand what work offered to them is about. A way to make the description of a task visible in the work lists is to use the global field __subject__. Listing 5 shows how this is specified in order to be displayed in the *subject* column in the work list (see the filled-in subject field for the first task in role-alpha’s work list in Figure 9). Besides the long-winded definition in Listing 5, a disadvantage of this way to name tasks is that during runtime, the __subject__ field (like any other field defined in a process) appears as a data field for each task and can be edited by a user during runtime (observe the data field in Figure 7). Filters can be used for explicitly defining the data available for every task, hence preventing the access and modification of the __subject__ value. However, this further complicates the code and is limited to the naming of sequential tasks only. As __subject__ is a global variable and its value (i.e. the name of a task) is re-entered prior to the commencement of a new task, it can not be used for giving parallel tasks individual names.

An example of the graphical representation of sequence process in Droflo is shown in the right hand side in the task execution window (Figure 7). This representation is the same for both version 1.0 of the process (Listing 4) and version 1.1 (Listing 5). Note that as the fields and attribute settings are not captured, it is not possible to show the names for the different tasks in the graphical model.

2.3 Enhydra Shark

Enhydra Shark is a Java workflow engine offering from Together Teamlösungen and ObjectWeb [7]. These entities sponsor a development collaboration platform at `http://enhydra.org` that accept contributions from developers worldwide. The workflow engine is one of several products under constant development. The system contains the following components (a sketch of its architecture, based on the WfMC reference model [5], is shown in Figure 10):

- a workflow engine called Shark;
- a Process Definition Tool called Together Workflow Editor (TWE) or JaWE
- a Management Console, TWS Admin, which is a Workflow Client as well as an Administrative and Monitoring Tool
- a set of ToolAgents (e.g. MailToolAgent, SOAPToolAgent, JavaScriptToolAgent, RuntimeApplicationToolAgent), which provide interfaces for different software, e.g., Javascript Interpreter.

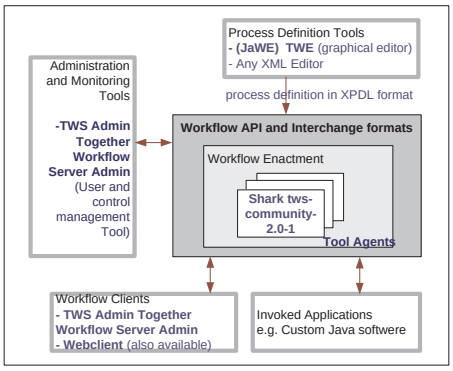


Figure 10 Architecture of Enhydra Shark

Enhydra Shark and JaWE require Java 1.4 version or higher. The main products considered in this review were the engine distributed through Shark TWS-community-2.0-1 (TWS - Together Workflow Server) and the editor JaWE TWE-community-2.2-1 (TWE - Together Workflow Editor). The evaluation was done through the TWS Admin client.

Together Workflow Editor (TWE) [3] uses XPDL 1.0 [28] as a process language. As such the ideas and constructs of XPDL are reflected in Enhydra. The main unit of organisation in XPDL is a *package*. A package contain one or several *process* definitions and can contain global *data* entities common for process definitions within the package. A process consists *activities* and *transitions* between them. Activities are assigned to *participants*. An activity may be performed *manually* or *automatically* and the participant may be either a human or an *application*. When an activity is completed according to the specification, the thread of control advances to the next activity. *Conditions* that constrain the transitions can be specified. When multiple threads are entering/exiting an activity, a *transition restriction* is specified for the activity, specifying whether it is an AND or an XOR *Join*/*Split*.

A process can be hierarchically structured through the use of *Block activities*. A block activity contains an *Activity set* which is a subprocesses following the same syntax and semantic rules as for processes. Furthermore, an activity can invoke a process which is specified independently. A special kind of activity is a *Routing activity*, which corresponds to a "dummy" activity used only for routing the control-flow within a process. In addition to the XML process definition, Enhydra Shark provides an in-house developed graphical notation where a process is visualised as a pool with work assignments to a participant represented by a "swim-lane". A screenshot of TWE is shown in Figure 11.

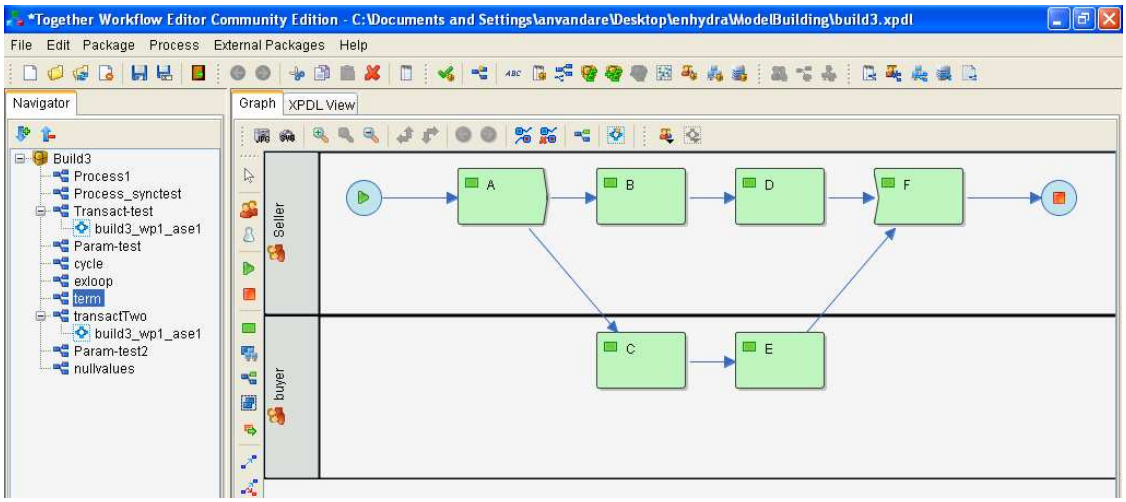


Figure 11 Together Workflow Editor (TWE)

As the model is built up in TWE all work is reflected in an XPDL file easily accessible and inspectable in the editor. The resulting specification of the process is saved as a file with the suffix .xpdl, which is an ordinary text file that may just as well, but less conveniently, be produced using any text editor.

The .xpdl-file with a process specification is loaded into Shark. Before a process is started, in the Administration Console the participants of a process definition are mapped to the actual users or user groups present in the system (for an example see Figure 12). At runtime the work items for an activity show in the worklist of the user(s) associated with the participant to which the activity was assigned at design time (for an example of a work list see Figure 13). A user starts the execution of a work item by selecting it from their list.

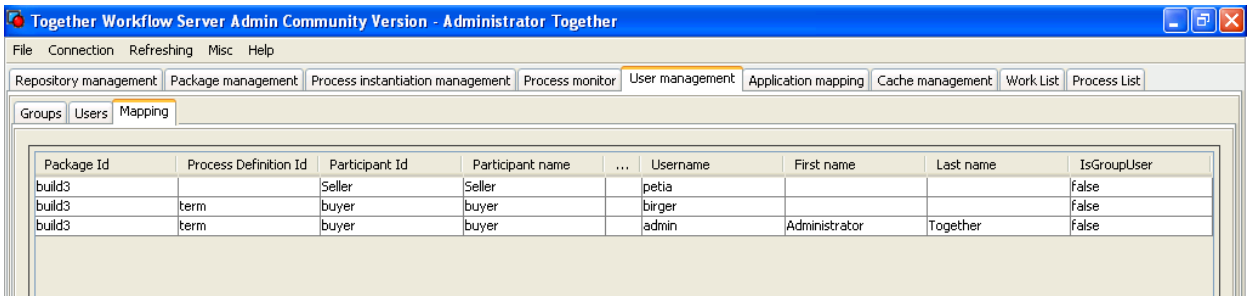


Figure 12 User-Participant Mapping in Enhydra Shark

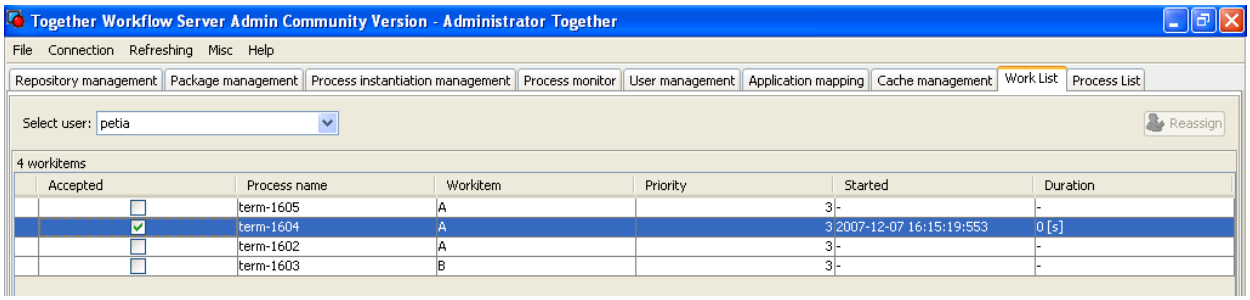


Figure 13 Work List in Enhydra Shark

3 Control-flow Patterns Evaluation

This section discusses the level of support offered by jBPM, OpenWFE and Enhydra Shark for the collection of control-flow patterns [25]. The control-flow patterns focus on the ordering of activities within a process. They are divided into the following groups: *Basic Control-flow Patterns* capturing elementary aspects of control-flow; *Advanced Branching and Synchronization Patterns* describing more complex branching and synchronization scenarios: *Iteration Patterns* describing various ways in which task repetition may be specified; *Termination Patterns* addressing the issue of when the execution of a workflow is considered to be finished; *Multiple Instances (MI) Patterns* delineating situations where there are multiple threads of execution in a workflow which relate to the same activity; *State-based Patterns* reflecting situations which are most easily modelled in workflow languages that support the notion of state; *Cancellation Patterns* categorizing the various cancellation scenarios that may be relevant for a workflow specification; and *Trigger Patterns* identifying the different triggering mechanisms appearing in a process context. The evaluation results for each of these groups are discussed below. Table 1 summarises the results of the complete set of evaluation for the three offerings using a three point scale where '+' corresponds to direct support for a pattern, '+/-' indicates that direct support is present but in a limited way, and '-' shows that the pattern is not directly supported. The evaluation criteria for assigning '+' or '+/-' are specified in [25].

3.1 Basic control-flow patterns

All three offerings provide direct support for the basic control flow patterns. Listings 6-8 illustrate how these patterns can be captured in OpenWFE. The constructs <sequence>, <concurrence> and <case> implement the behaviours

Basic Control–flow	1	2	3	Termination	1	2	3
1. Sequence	+	+	+	11. Implicit Termination	+	+	+
2. Parallel Split	+	+	+	43. Explicit Termination	–	–	–
3. Synchronization	+	+	+	Multiple Instances			
4. Exclusive Choice	+	+	+	12. MI without Synchronization	+	+	+
5. Simple Merge	+	+	+	13. MI with a priori Design Time Knl.	–	+	–
Advanced Synchronization				14. MI with a priori Runtime Knl.	–	+	–
6. Multiple Choice	–	+/–	+	15. MI without a priori Runtime Knl.	–	–	–
7. Str Synchronizing Merge	–	–	–	27. Complete MI Activity	–	–	–
8. Multiple Merge	+	–	–	34. Static Partial Join for MI	–	+	–
9. Structured Discriminator	–	+	–	35. Static Canc. Partial Join for MI	–	+	–
28. Blocking Discriminator	–	–	–	36. Dynamic Partial Join for MI	–	–	–
29. Cancelling Discriminator	–	+	–	State-Based			
30. Structured Partial Join	–	+	–	16. Deferred Choice	+	–	–
31. Blocking Partial Join	–	–	–	39. Critical Section	–	–	–
32. Cancelling Partial Join	–	+	–	17. Interleaved Parallel Routing	–	+/–	–
33. Generalized AND-Join	+	–	–	40. Interleaved Routing	–	+	–
37. Local Sync. Merge	–	+/–	–	18. Milestone	–	–	–
38. General Sync. Merge	–	–	–	Cancellation			
41. Thread Merge	+/–	–	–	19. Cancel Activity	+	–	–
42. Thread Split	+/–	–	–	20. Cancel Case	–	+/–	+
Iteration				25. Cancel Region	–	–	–
10. Arbitrary Cycles	+	+	+	26. Cancel MI Activity	–	–	–
21. Structured Loop	–	+	–	Trigger			
22. Recursion	–	+	+	23. Transient Trigger	+	+	–
				24. Persistent Trigger	–	–	–

Table 1. Support for the Control-flow Patterns in **1**–JBoss jBPM 3.1.4, **2**–OpenWFE 1.7.3, and **3**–Enhydra Shark 2.0

for the Sequence, Parallel Split and Exclusive Choice patterns respectively. As the language is block structured, the closing tags </concurrence> and </case> provide the semantics for the Synchronization and Simple Merge patterns.

Listing 6 WCP-1 Sequence in OpenWFE

```
<sequence>
    <participant ref="role-alpha"
       description="design"/>
    <participant ref="role-bravo"
       description="implement"/>
    <participant ref="role-charly"
       description="deploy"/>
</sequence>
```

Listing 7 WCP-2 Parallel Split and CPF3 Synchronization in OpenWFE

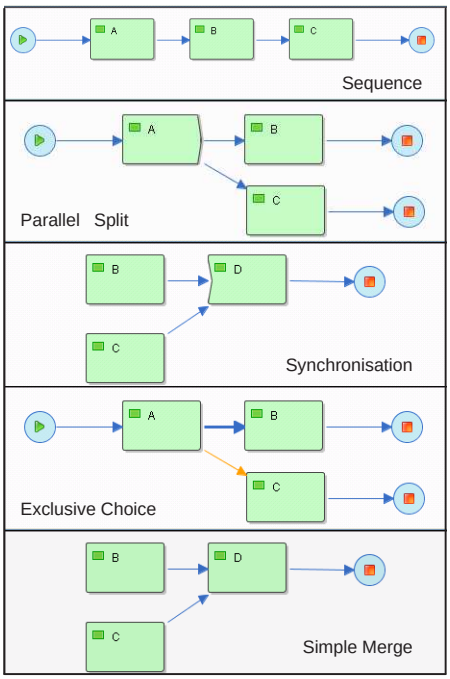
```
<concurrence>
    <participant ref="role-alpha"
       description="implement"/>
    <participant ref="role-bravo"
       description="test"/>
    <participant ref="role-charly"
       description="document"/>
</concurrence>
```

Listing 8 WCP-4 Exclusive Choice and WCP-5 Simple Merge in OpenWFE

```
<set field="alternative" type="string"
   value="select task: 1, 2 or 3" />
<participant ref="role-alpha"
   description="select task"/>
<case>
   <equals field-value="alternative"
         other-value="1"/>
     <participant ref="role-alpha"
        description="deploy"/>
   <equals field-value="alternative"
         other-value="2"/>
     <participant ref="role-bravo"
        description="correct"/>
   <equals field-value="alternative"
         other-value="3"/>
     <participant ref="refactor"
        description="task3"/>
</case>
```

Enhydra Shark captures Sequences through a *transition* between activities (see Figure 14 and Listing 9). A Parallel Split (Listing 10) is captured through an *activity* node with the *transition restriction* <**Split** Type="**AND**">. Synchronization (Listing 11) is captured in the same way, but the transition restriction specified is <**Join** Type="**AND**">. Exclusive Choice and Simple Merge are captured as Parallel Split and Synchronization, respectively, but with **XOR** (instead of AND) specified as their transition restriction. For the Exclusive Choice *transition conditions* are also specified on the transitions leaving the XOR activity node. Graphically, transitions with transition conditions specified on them are shown by a bold arrow. It is possible to specify "Otherwise" as a transition condition on a transition (which will evaluate to true if the preceding conditions do not): this is graphically denoted by an orange coloured arrow.

Figure 14 Basic Patterns in Enhydra Shark



Listing 9 WCP-1 Sequence in Enhydra Shark

```
<Activity Id="A" Name="A">..</Activity>
<Activity Id="B" Name="B">..</Activity>
 ...
 <Transition From="A" Id="Tr1".. To="B"/>
```

Listing 10 WCP-2 Parallel Split in Enhydra Shark

```
<Activity Id="A" Name="A"> ..
  <TransitionRestriction>
    <Split Type="AND"> ..
         <TransitionRef Id="Tr1"/>
         <TransitionRef Id="Tr2"/>
       ...
</Activity>
<Activity Id="B" Name="B"></Activity>
<Activity Id="C" Name="C"></Activity>
...
  <Transition From="A" Id="Tr1".. To="B"/>
  <Transition From="A" Id="Tr2".. To="C"/>
```

Listing 11 WCP-3 Synchronization in Enhydra Shark

```
  <Activity Id="B" Name="B">..</Activity>
  <Activity Id="C" Name="C">..</Activity>
  <Activity Id="D" Name="D"> ..
      <TransitionRestriction>
        <Join Type="AND"/>
    ...
  <Transition From="B" Id="Tr3" .. To="D"/>
  <Transition From="C" Id="Tr4" .. To="D"/>
```

Listing 12 WCP-1 Sequence in jBPM

```
<task-node name="task1">
  <task name="task1a" swimlane="ernie">
  <transition name="" to="task2">
</task-node>
<task-node name="task2">
  <task name="task2a" swimlane="ernie">
  <transition name="" to="end1">
</task-node>
```

Listing 13 WCP-2 Parallel Split in jBPM

```
<fork name="AND-split">
  <transition name="trA" to="task1A">
  <transition name="trB" to="task1B">
</fork>
<task-node name="task1A">
  <task name="task1A" swimlane="ernie">
  <transition name="" to="end1">
</task-node>
<task-node name="task1B">
  <task name="task1B" swimlane="ernie">
  <transition name="" to="end1">
</task-node>
```

Listing 14 WCP-2 Parallel Split in jBPM, alt2

```
<task-node name="task1-node">
      <task name="task1A" swimlane="ernie">
      <task name="task1B" swimlane="ernie">
      <transition name="" to="end1">
</task-node>
```

Listing 15 WCP-3 Synchronization in jBPM

```
<fork name="fork1">
  <transition name="tr1" to="task1A-node">
  <transition name="tr2" to="task1B-node">
</fork>
<task-node name="task1A-node">
  <task name="task1A" swimlane="ernie">
  <transition name="" to="join1">
</task-node>
<task-node name="task1B-node">
  <task name="task1B" swimlane="ernie">
  <transition name="" to="join1">
</task-node>
<join name="join1">
  <transition name="" to="task2-node">
</join>
```

Listing 16 WCP-4 Exclusive Choice in jBPM

```
<decision name="xor">
  <transition name="urgent" to="task-b">
    <condition>scenario==1</condition>
  </transition>
  <transition name="dont care" to="task-c">
    <condition>scenario==2</condition>
  </transition>
  <transition name="forget about it" to="task-d">
    <condition>scenario!=1
         and scenario!=2</condition>
  </transition>
</decision>
```

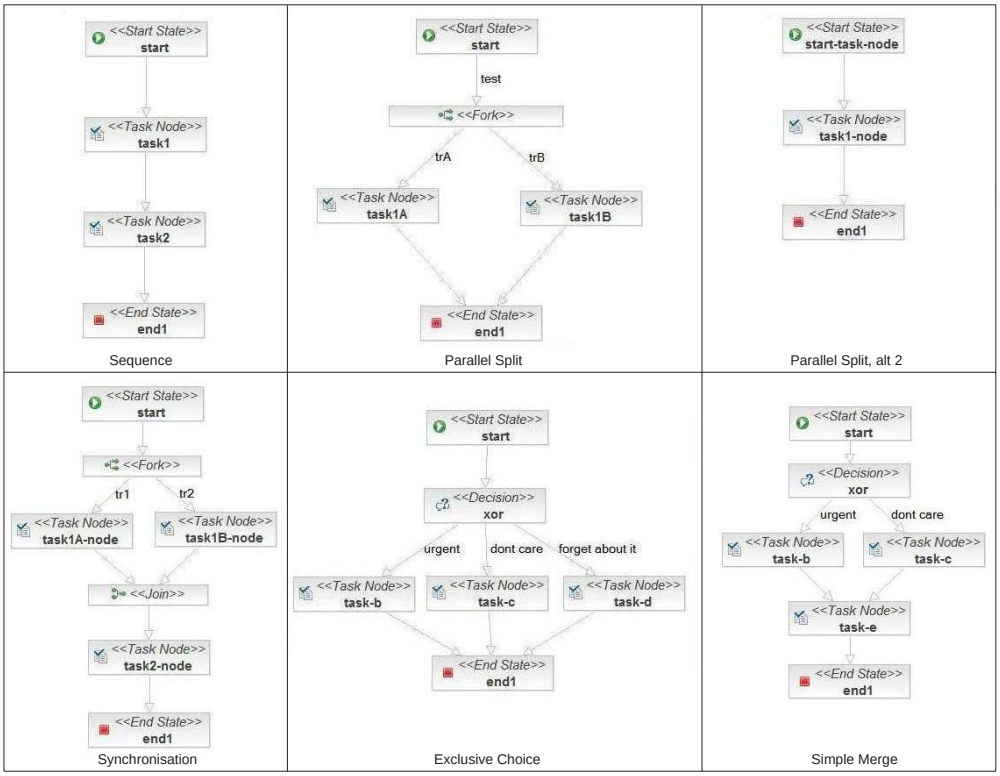


Figure 15 Basic Patterns in jBPM

Listings 12-17 show how the basic patterns can be captured in jBPM. In Figure 15, the corresponding graphical representations for jBPM are shown. A Sequence is represented by a *transition* between two *task-nodes*, the Parallel Split pattern is captured through a *fork node* with a number of outgoing transitions (Listing 13). An alternative solution is to specify the *tasks* to be run in parallel within the same task-node (Listing 14)[7]. Synchronization is captured through a *join node* with a number of incoming transitions (Listing 15). The Exclusive Choice pattern is captured by specifying *condition*s on the transitions leaving a *decision node* (Listing 16). The first expression that evaluates to true results in the continuation of the flow of control in the transition with which the expression is associated. The Simple Merge pattern is captured by a task-node which joins the transitions emanating from the set of task-nodes from which the thread(s) of control should be merged (Listing 17).

Listing 17 WCP-5 Simple Merge in jBPM

```
<decision name="xor">
   <transition name="urgent" to="task-b">
      <condition expression=.../>
   </transition>
   <transition name="dont care" to="task-c">
      <condition expression=.../>
   </transition>
</decision>
<task-node name="task-b">
   <task name="task-b" swimlane="ernie">
      <transition name="" to="task-e">
</task-node>
<task-node name="task-c">
   <task name="task-c" swimlane="ernie">
      <transition name="" to="task-e">
</task-node>
<task-node name="task-e">
   <task name="task-e" swimlane="ernie">
   <transition name="" to="end1">
</task-node>
```

Listing 18 WCP-8 Multi Merge in jBPM

```
<fork name="fork1">
  <transition name="tr1" to="task1A-node">
  <transition name="tr2" to="task1B-node">
</fork>
<task-node name="task1A-node">
  <task name="task1A" swimlane="ernie">
  <transition name="" to="task2-node">
</task-node>
<task-node name="task1B-node">
  <task name="task1B" swimlane="ernie">
  <transition name="" to="task2-node">
</task-node>
<task-node name="task2-node">
   <task name="task2" swimlane="ernie">
   <transition name="" to="end1">
</task-node>
```

[7] Note that the graphical representation for the second solution for the Parallel Split pattern is similar to the graphical representation of the Sequence pattern, which may lead to confusion for some users.

3.2 Advanced branching and synchronization patterns

The support for the patterns in this group is limited. jBPM provides support for the Multiple Merge and the Generalized AND-Join patterns and partial support for the Thread Merge and the Thread Split patterns (it is rated a '–' for the Structured Synchronizing Merge as the context condition of that pattern requires the presence of a corresponding Multiple Choice which is not supported by jBPM). If, in the bottom righthand model of Figure 15, the decision node is replaced by a fork node, the model will capture the semantics of the Multiple Merge pattern. The jPDL definition for this pattern is given in Listing 18. Note that as a consequence of the use of the fork, it is guaranteed that the last node, i.e. "task2-node" will be executed twice, which distinguishes this pattern from the Simple Merge pattern. Thread Merge and Thread Split can be achieved programmatically in Java through defining two new Nodes implementing the corresponding behaviour (according to the evaluation criteria of these patterns use of programmatic extension for achieving their behaviour rates as partial support).

The support for the Generalized AND-join in jBPM is demonstrated by the lefthand model in Figure 16. The join node in this model is initiated twice, capturing correctly the Generalized AND-join semantics. In this model task-C acts as a Multiple Merge and is executed twice (as both incoming branches are enabled separately) resulting in two concurrent threads of execution at the second fork node (each of which results in a thread of execution in both of the outgoing branches from the Fork node). The following join node can only execute when it has received the thread of control on each of its incoming branches. As there are two threads of control flowing down each of the incoming branches to the join node (as a consequence of the fork node firing twice), it is possible that more than one thread of control will be received on one input branch to the join before a thread of control is received on the other and the join is able to fire. Where this occurs, the join keeps track of the number of execution threads received on each incoming branch and it only executes when one has been received at every incoming branch. Any additional control threads are retained for future firings, hence the join will ultimately execute twice and two execution threads are passed onto task-F.

The righthand model in Figure 16 illustrates the scenario with which jBPM's support for the Local Synchronizing Merge was tested. The join node in this model was intended to act as a local synchronising merge by keeping track of whether additional execution threads can still be received on input branches that have not yet been enabled. For example, after execution of the nodes taskA, taskB and task-d, the join would wait as it is possible that task-b will be executed. If task-a is chosen rather than task-b, the join can continue and trigger the execution of task-bd, if not, it will await the completion of task-b and then start the execution of task-bd. However, while syntactically correct, this process get stalled in the cases when only one of the tasks task-b and task-d gets selected and executed as the Join node keeps waiting for the completion of both.

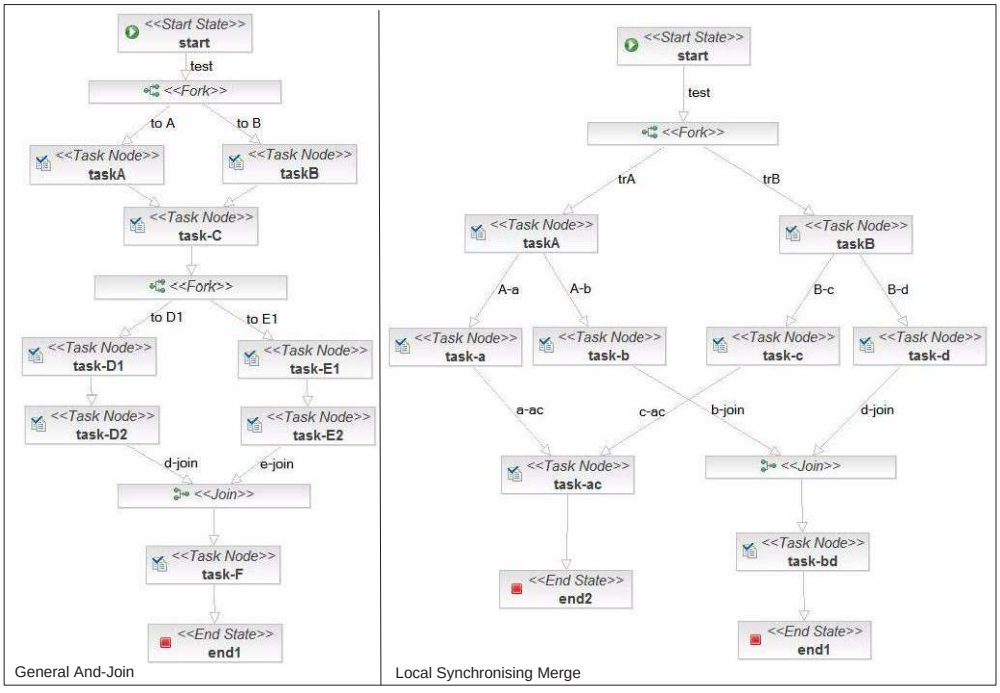


Figure 16 Patterns WCP-33 and WCP-37 in jBPM

Listing 19 WCP-9 Structured Discriminator in OpenWFE

```
<concurrence
    count="1"
    remaining="forget">
        <participant ref="role-alpha"
           description="mail contact"/>
        <participant ref="role-bravo"
           description="email contact"/>
        <participant ref="role-charly"
           description="sms contact"/>/>
</concurrence>
```

Listing 20 WCP-32 Cancelling Partial Join in OpenWFE

```
<concurrence
    count="2"
    remaining="cancel">
        <participant ref="role-alpha"
           description="mail contact"/>
        <participant ref="role-bravo"
           description="email contact"/>
        <participant ref="role-charly"
           description="sms contact"/>/>
</concurrence>
```

From the advanced branching and synchronization patterns, OpenWFE provides direct support for the Structured and Cancelling variants of the Discriminator and Partial Join patterns. Their behaviour is achieved through the use of the <concurrence> construct with attribute settings that allow for the specification of the number of incoming branches that must receive an execution thread in order for the join construct to execute (i.e. the *count* attribute) and also for the specification of what should happen to the other (incoming) branches once the join has executed (the *remaining* attribute which can be set to 'forget' or to 'cancel' remaining execution threads received in these incoming branches). Listing 19 illustrates OpenWFE's support for the Structured Discriminator pattern. At runtime three work items are started in parallel, the first one to complete (indicated by the count="1" setting) triggers the remainder of the flow (not shown) and the remaining work items are allowed to complete although this does not influence the flow in any way (due to the remaining="forget" setting). By changing the settings for the attributes 'count' and 'remaining' the other forms of Discriminator and Partial Join patterns are also easily captured. Listing 20 shows an example for the Cancelling Partial Join.

OpenWFE also provides partial support for the Multiple Choice pattern. It is achieved through a collection of if-then clauses executing concurrently, where several of the if-conditions may evaluate to true. Due to the structured nature of workflow specifications in OpenWFE there is no support for patterns such as the Multiple Merge and the Generalized AND-join. As with jBPM, the Structured Synchronizing merge is not considered to be supported in OpenWFE due to the absence of a dedicated Multiple Choice construct. The Local Synchronizing Merge is partially supported as the pattern can only be captured in structured scenarios[8], while general scenarios such as the one presented for jBPM (in the righthand model in Figure 16) can not be captured in OpenWFE.

In this category, Enhydra Shark only provides support for the Multiple Choice pattern, which similar to OpenWFE, is captured by defining overlapping conditions on transitions emanating from an AND-split activity (see Figure 17 and Listing 21[9]). Enhydra Shark also proposes the model in Figure 18a, as a solution for the Multiple Merge pattern. In this model activity A is a multi-choice activity after which one or several of activities: B, C, and/or D can be started. At completion each one of these activities asynchronously invokes an instance of the subprocess S1. However, this is not considered to fully capture the semantics for the Multiple Choice pattern, as general scenarios like the one shown in Figure 18b can not be captured in this manner. The same reasoning applies for OpenWFE (which also provides the possibility to invoke a subflow to run asynchronously to the main process).

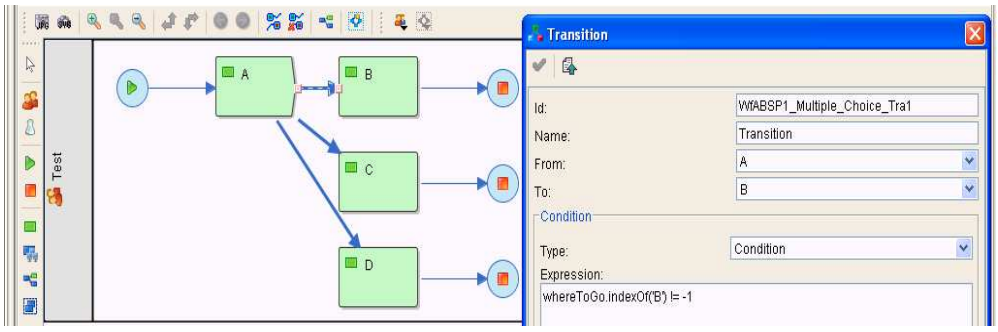


Figure 17 WCP-6 Multiple Choice in Enhydra Shark

[8] An example of such a scenario is presented in the animation for the pattern available at `www.workflowpatterns.com/patterns/control/new/wcp37_animation.php`

[9] For space reasons the transition Id "WfABSP1_Multiple_Choice_Tra1" in Figure 17 is abbreviated to "Tr1" in Listing 21 and the variable "whereToGo" is replaced with "X".

Listing 21 WCP-6 Multiple Choice in Enhydra Shark

```
<Activity Id="A" Name="A">
  <TransitionRestrictions>
    <TransitionRestriction>
      <Split Type="AND">
         <TransitionRefs>
            <TransitionRef Id="Tr1"/>
            <TransitionRef Id="Tr2"/>
            <TransitionRef Id="Tr3"/>
         </TransitionRefs>
      </Split>
    </TransitionRestriction>
  </TransitionRestrictions>
  <ExtendedAttributes>
     <ExtendedAttribute Name="VariableToProcess_
     UPDATE" Value="X"/>
  </ExtendedAttributes>
</Activity>
 <Activity Id="B" Name="B"></Activity>
 <Activity Id="C" Name="C"></Activity>
 <Activity Id="D" Name="D"></Activity>
...
 <Transition From="A" Id="Tr1" .." To="B">
    <Condition Type="CONDITION">X.indexOf('B')!= -1
    </Condition>
 </Transition>
 <Transition From="A" Id="Tr2" .. To="C">
    <Condition Type="CONDITION">X.indexOf('C')!= -1
    </Condition>
 </Transition>
 <Transition From="A" Id="Tr3" .. To="D">
    <Condition Type="CONDITION">X.indexOf('D')!= -1
    </Condition>
 </Transition>
...
```

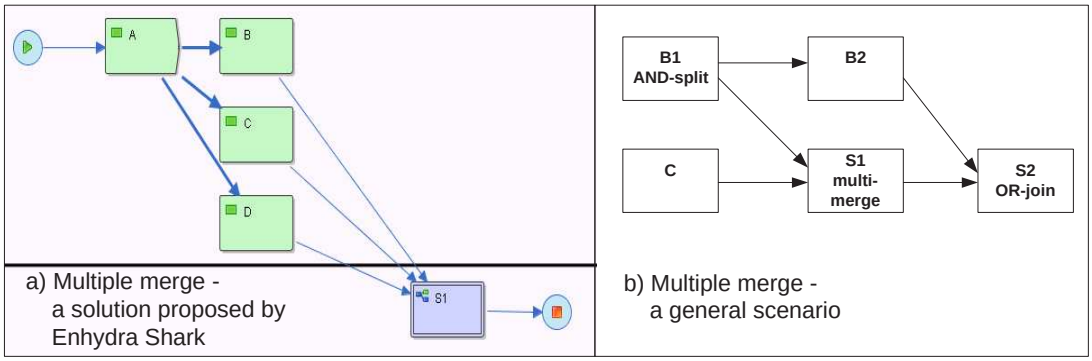
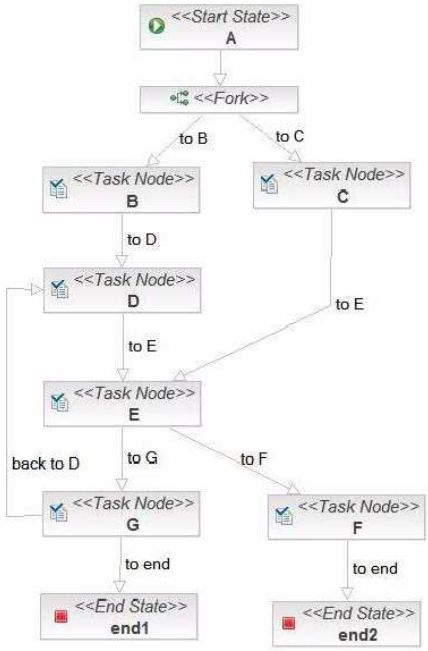


Figure 18 WCP-7 Multiple Merge proposed in an example contained in Enhydra Shark's default installation

Listing 22 CPF-10 Arbitrary Cycles in jBPM

```
<start-state name="A">
      <task name="A" swimlane="ernie">
      <transition name="" to="fork1">
   </start-state>
   <end-state name="end1">
   <task-node name="B">
      <task name="B" swimlane="ernie">
      <transition name="to D" to="D">
   </task-node>
   <task-node name="D">
      <task name="D" swimlane="ernie">
      <transition name="to E" to="E">
   </task-node>
   <task-node name="E">
      <task name="E" swimlane="ernie">
      <transition name="to G" to="G">
      <transition name="to F" to="F">
   </task-node>
   <task-node name="G">
      <task name="G" swimlane="ernie">
      <transition name="to end" to="end1">
      <transition name="back to D" to="D">
   </task-node>
   <fork name="fork1">
      <transition name="to B" to="B">
      <transition name="to C" to="C">
   </fork>
   <task-node name="C">
      <task name="C" swimlane="ernie">
      <transition name="to E" to="E">
   </task-node>
   <end-state name="end2">
   <task-node name="F">
      <task name="F" swimlane="ernie">
      <transition name="to end" to="end2">
   </task-node>
```

Figure 19 WCP-10 Arbitrary Cycles, graphical representation of the model in Listing 22



Listing 23 WCP-21 while-do loop in OpenWFE, alt1

```
<loop>
   <until> condition</until>
   <sequence>iteration body</sequence>
</loop>
```

Listing 24 WCP-21 while-do loop in OpenWFE, alt2

```
<loop>
   <while> condition </while>
   <sequence>iteration body</sequence>
</loop>
```

Listing 25 WCP-21 repeat-until loop in OpenWFE, alt1

```
<loop>
   <sequence>iteration body</sequence>
   <until>condition</until>
</loop>
```

Listing 26 WCP-21 repeat-until loop in OpenWFE, alt2

```
<loop>
   <sequence> iteration body</sequence>
   <while> condition </while>
</loop>
```

3.3 Iteration patterns

In terms of the iteration patterns, jBPM directly supports the Arbitrary Cycles pattern, but does not have dedicated constructs for structured loops nor does it support recursion[10]. jBPM's support for the Arbitrary Cycles pattern is shown in Listing 22 and in Figure 19. The arbitrary cycle shown has two entry points (one through task-node B, the other through task-node E) and two exit points (one after task-node E and one after task-node G).

OpenWFE provides direct support for the Structured Loop and Recursion patterns. Both the while-do and repeat-until variants of the Structured Loop pattern are supported by the <loop> construct. A <loop> construct contains a *loop condition* specified through the use of <while> or <until> clauses and an *iteration body* enclosed within a <sequence> clause. The loop condition (i.e. the <while> and <until> parts) can appear before or after the iteration body (i.e. the <sequence> specification) which determines whether the loop is of type while-do or repeat-until (see listings 23- 26 for illustrations of these solutions).

Furthermore, the <iterator> construct can be used to assemble loops. The <iterator> implements the behaviour commonly associated with for-loops from classical programming languages, where an index variable takes values across a specified range on a sequential basis and the body of the loop is executed once for each possible value. Listing 28[11] shows an example of an <iterator> construct in OpenWFE.

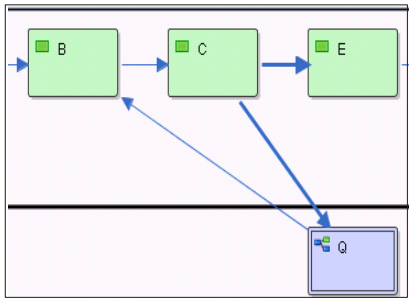
The cursor concept in OpenWFE allows for the thread of control in a process instance to be moved back and forth. Consider for example Listing 29, at runtime the three activities can be executed in sequence, but at any point in time, users with sufficient privileges to execute these work items can advance or go back one or more steps in the flow and can even choose to exit the flow before the sequence is finished. The commands "back", "skip", "break" (or "cancel"), and "rewind" (moving back to the beginning of the cursor), are used for these purposes. These commands can either be specified at runtime in a field __cursor_command__ (which appears by default in the user interface during the execution of a task specified within a cursor), or at design time as exemplified in Listing 30. It is possible to limit the use of these commands by specifying them as disallowed in the definition of a given cursor, e.g. <cursor disallow="rewind, break">. Consequently, the cursor command can also be used for specifying arbitrary cycles with multiple exit points, thus fulfilling the criteria for full support of this pattern. Note, however that the cursor construct can not be used for capturing a scenario such as the one illustrated for jBPM in Figure 19 where the cycle can not only be exited at different points but also additional threads of control can enter the cycle at various places. In the new Ruby version of the tool, a *jump* construct is introduced [23] which seems to act as a GO-TO operator and thus enables the modelling of arbitrary cycles with multiple entry points.

In terms of the iteration patterns, Enhydra Shark, similar to jBPM, directly supports the Arbitrary Cycles pattern (allowing cycles both with multiple entry and exit points), but does not have dedicated constructs for structured loops. Enhydra Shark also supports recursion. A workflow can invoke itself by means of subflow. For an example of this, see Figure 20 and Listing 27.

[10] In jBPM specification [13], Chapter 9.6 Superstates, it is stated that Super States can be nested recursively. However, in the graphical process designer (GPD) it is currently not possible to specify the components of a Super State, hence we consider recursion as not yet being supported.

[11] With some small modifications, this solution is taken from Chapter 7 Workflow patterns, Pattern 14, in the OpenWFE specification [20].

Figure 20 WCP-22 Recursion in Enhydra Shark, graphical representation of the model in Listing 22



Listing 27 CPF-22 Recursion in Enhydra Shark

```
<WorkflowProcess .. Id="cycle" ..>
  ..
    <Activity Id="B" Name="B">
       .. <Join Type="XOR"/> .. </Activity>
    <Activity Id="C" Name="C">
        .. <Split Type="XOR"> .. </Split>
        .. </Activity>
    <Activity Id="E" Name="E"></Activity>
    <Activity Id="Q">
        <Implementation>
            <SubFlow Id="cycle"/>
        </Implementation>
    </Activity>
  ..
</WorkflowProcess>
```

Listing 28 WCP-21 For-loop in OpenWFE

```
<iterator
   on-value="alpha,bravo,charly"
   to-field="rolename">
   <sequence>
      <participant ref="role-${f:rolename}"/>
   </sequence>
</iterator>
```

Listing 29 The cursor construct in OpenWFE

```
<cursor>
  <participant ref="role-alpha" description=
     "drug experiment on animals"/>
  <participant ref="role-bravo" description=
     "small human sample experiment"/>
  <participant ref="role-charly" description=
     "large human sample experiment"/>
</cursor>
```

Listing 30 The cursor construct in OpenWFE, design time established behaviour

```
<cursor>
  <set field="mark" type="integer"/>
  <participant ref="role-alpha"
     description="exam"/>
  <if>
    <greater-than field-value="mark"
                  other-value="5"/>
    <break />
  </if>
  <participant ref="role-bravo"
     description="study"/>
  <participant ref="role-alpha"
     description="complement"/>
  <if>
    <lesser-than field-value="mark"
       other-value="3"/>
    <back step="3"/>
  </if>
</cursor>
```

3.4 Termination patterns

All three offerings support the Implicit Termination pattern, i.e. a process instance completes when there is not any work item left to be done (i.e. there is no need to specify a single explicit end node). While jBPM notifies the user executing the last work item about the completion of a process, Enhydra Shark and OpenWFE do not. Currently, OpenWFE only identifies the completion of process instances in the log of the execution history. In Enhydra Shark, a completed task is displayed in the Process List as having a "closed.completed" status.

3.5 Multiple instances patterns

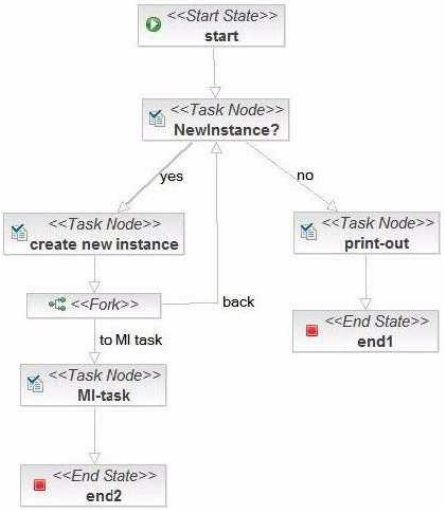
There is virtually no support for multiple instance tasks in jBPM or Enhydra Shark. Direct support is only provided for the Multiple Instances without Synchronization pattern as new instances of tasks can be spawned off through the use of a loop. This is illustrated for jBPM with the model shown in Figure 21 where as part of the loop based on the *NewInstance?* task-node, new instances of the task-node MI-task are spawned off. As Enhydra Shark is block structured, in every iteration of such a loop an instance of a separate subprocess is invoked to run asynchronously with the main-process.

Similarly to Enhydra Shark, the Multiple Instances without Synchronization pattern is achieved in OpenWFE by launching new instances as subprocesses through a loop. Setting the *forget* attribute to "true" ensures that synchronization over the created instances will not take place. This solution is illustrated in Listing 31. In this example three instances of the subprocess "run test" are spawned-off through the loop. The number '3' of instances to be created is decided at design time and is hard coded in the loop condition.

In OpenWFE the Multiple Instances with a priori Design Time Knowledge and Multiple Instances with a priori Runtime Knowledge patterns are captured through the <concurrent-iterator> construct through which a number of instances (determined during design or runtime) are spawned off. At completion, the instances are synchronised. Listing 32 shows how three instances of a task "test" can be created that run concurrently. Each of this instances will have a unique number in its *index* field (1, 2 or 3 in this example) which is one of the items in the "on-value" list. At completion, the instances will be synchronised (through </concurrent-iterator>). Listing 33 shows how multiple instances can be created on the basis of the values in a list, called "iteration_list" in this example. These values will be

retrieved during the execution of the task “distribute tests to subjects”, i.e. at runtime but before the commencement of the multiple instances task “run test”. In “run test” a work item will be created for every item in the iteration-list. The variable “user” acts as an index variable on the basis of which each work item will be created. At completion the created work items will be synchronised (through </concurrent-iterator>) and the task “summarize results” will be enabled. Note that the work items of the multiple instance task “run test” are distributed to the participants included in the “iteration_list” (which is passed to distinct instances through the variable “user”, as illustrated in lines 8-9 in the listing). This is an example of the Deferred Allocation pattern from the Resource Perspective which will be further elaborated upon in Section 5.

Figure 21 WCP-12 MI without Synchronization in jBPM



Listing 31 WCP-12 MI without Synchronization in OpenWFE

```
<sequence> <participant ref="role-a"
    description="prepare first set of tests"/>
  <loop>
    <sequence>
      <subprocess ref="run test" forget="true"/>
        <inc variable="idx0" />
    </sequence>
    <while>
      <lesser-than variable-value="idx0"
      other-value="3" />
    </while>
  </loop>
  <participant ref="role-bravo"
    description="report on the commencement
    of the tests"/>
</sequence>
<process-definition name="run test">
    <sequence>
        <participant ref="role-charly"
           description="test"/>
    </sequence>
</process-definition>
```

Listing 32 CPF-13 MI with a priory Design Time Knowledge in OpenWFE

```
<concurrent-iterator on-value="1, 2, 3"
    to-field="index">
  <participant ref="role-charly"
     description="test"/>
</concurrent-iterator>
<participant ref="role-alpha"
     description"summarize results"/>
```

Listing 33 CPF-14 MI with a priory Runtime Knowledge in OpenWFE

```
<sequence>
  <set field="iteration_list"
     type="StringMapAttribute"
     value="type users: alpha,bravo.."/>
  <participant ref="role-alpha"
     description="distribute tests to subjects"/>
  <concurrent-iterator on-field-value="iteration_list"
     to-field="user">
     <participant ref="role-${f:user}"
        description="run test"/>
  </concurrent-iterator>
  <participant ref="role-alpha"
    description="summarize results"/>
</sequence>
```

Listing 34 WCP-34 Static Partial Join for MI in OpenWFE

```
<sequence>
  <set field="nr_of_inst"
     value="1,2,3" />
  <set field="nr_to_be_completed"
     type="Integer" value="" />
  <participant ref="role-alpha"
     description="set up test"/>
  <concurrent-iterator
        on-value="${f:nr_of_inst}"
        to-field="index"
        count="${f:nr_to_be_completed}"
        remaining="forget"
        merge="first"
        merge-type="mix">
     <participant ref="role-bravo"
        description="run test"/>
  </concurrent-iterator>
  <participant ref="role-alpha"
     description="summarize test results"/>
</sequence>
```

The Multiple Instances without a Priori Runtime Knowledge pattern (i.e. WCP-15), which is one of the more advanced multiple instances patterns, is not supported in any of the offerings. The distinguishing feature of this pattern is that new instances of the multiple instances task can be created after the task has commenced. The concurrent-iterator construct in OpenWFE can only be used when the number of instances to be created is known before the commencement of the multiple instance task. The repeated invocation of a subprocess within a loop with the attribute forget=“true”[12] does not synchronize the created instances and hence does not capture the full semantics of the pattern.

Listing 34 shows how the *count* attribute of the <concurrent-iterator> construct can be used to achieve a partial join (note that in this listing a variable called “nr_to_be_completed” is used at runtime to determine the number of threads that need to be synchronized and merged). The Static Partial Join for Multiple Instances pattern is supported by setting the *remaining* attribute to “forget” so that threads remaining after the completion of the multiple instance task are not cancelled. The Cancelling Partial Join for Multiple Instances pattern is implemented by setting the *remaining* attribute to “cancel”. There is no dedicated support for cancelling (i.e. WCP-26) or force-completing multiple instance

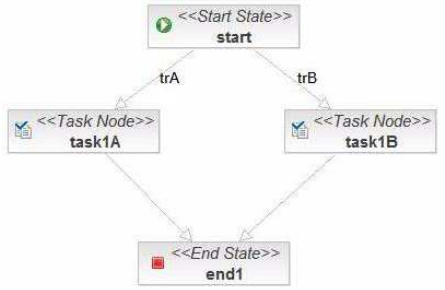
[12] This is the solution proposed by OpenWFE in [20] (Chapter 7, Workflow Patterns, Pattern 15)

tasks (WCP-27) in OpenWFE, nor for dynamically creating new instances of a multiple instance task after it has started and for disabling that ability during execution (WCP-36).

3.6 State-based patterns

From the State-based patterns, jBPM only provides support for the Deferred Choice pattern. This is illustrated in Figure 22 and in Listing 35. Surprisingly, this pattern is not supported in OpenWFE. According to the self-evaluation of OpenWFE ([20], Chapter 7 Workflow Patterns, WP16) the code in Listing 19 with the *remaining* attribute set to "cancel" is proposed as a solution for the deferred choice. However, as discussed earlier, this is a solution for the Cancelling Discriminator and not the Deferred Choice pattern as all branches are started and the first to complete cancels the others whereas in a deferred choice the work items associated with the first of the tasks in each of the alternate branches associated with the deferred choice are offered and the first one to be chosen leads to the withdrawal of the others.

Figure 22 WCP-16 Deferred Choice in jBPM



Listing 35 WCP-16 Deferred Choice in jPDL

```
1   <start-state name="start">
2      <task name="start-task" swimlane="ernie">
3      <transition name="trA" to="task1A">
4      <transition name="trB" to="task1B">
5   </start-state>
6   <end-state name="end1">
7   <task-node name="task1A">
8      <task name="task1A" swimlane="ernie">
9      <transition name="" to="end1">
10  </task-node>
11  <task-node name="task1B">
12     <task name="task1B" swimlane="ernie">
13     <transition name="" to="end1">
14  </task-node>
```

Listing 36 WCP-40 Interleaved Routing in OpenWFE

```
1 <interleaved>
2    <participant ref="role-alpha"
3       description="take exam A"/>
4    <participant ref="role-alpha"
5       description="take exam B"/>
6    <participant ref="role-alpha"
7       description="take exam C"/>
8 </interleaved>
```

Listing 37 WCP-17 Interleaved Parallel Routing in OpenWFE

```
1  <interleaved>
2     <participant ref="role-alpha"
3        description="take subsidiary subject"/>
4     <sequence>
5        <participant ref="role-alpha"
6           description="take methodology unit"/>
7        <participant ref="role-alpha"
8           description="write thesis"/>
9     </sequence>
10 </interleaved>
```

The <interleaved> construct in OpenWFE provides direct support for Interleaved Routing, as illustrated in Listing 36. At runtime work items corresponding to the three tasks are executed in arbitrary order and not concurrently. This construct however provides only partial support for Interleaved Parallel Routing as sequences of tasks (that need to be preserved) cannot be interrupted and have to be completed first before other activities (or other sequences of activities) can take their turn. In terms of Listing 37 this means that a work item of the task "take subsidiary subject" cannot be executed in between work items of the tasks "take methodology unit" and "write thesis". The Critical Section pattern is not supported in OpenWFE as the sub-threads constituting the critical sections would have to be placed in one interleaved block thus limiting the use of the pattern in unstructured flows. Also the Milestone pattern lacks support in OpenWFE.

Enhydra Shark does not provide any support for the State-based patterns.

3.7 Cancellation patterns

From the cancellation patterns only Cancel Case is supported in OpenWFE. Cancellation is specified at design time with the expression <cancel-process/>. However, any subprocesses invoked and running when <cancel-process/> is reached do not get cancelled. Therefore, this pattern is only considered to be partially supported. In practice, cancellation of an activity or a case can be done by a workflow administrator through the command-line control program, but as this form of cancellation is an error-handling measure rather than a normal process activity, it is not considered to provide support for the cancellation patterns.

In jBPM a task or a process can be forcibly ended in runtime by a user (not necessarily an administrator) through the Monitoring mode (see Figure 3). A forcibly ended task (through the "End" command) remains in the execution log, while a forcibly ended process (through the "Delete" command) is removed from the execution log. Therefore the Cancel Activity pattern is considered to be supported, while the Cancel Case pattern is not considered to be supported.

Enhydra Shark supports cancel case at runtime. A case can be cancelled trough the administration console by selecting “Terminate” for a specific instance. The case’s status is logged as “closed.terminated”. None of the other cancellation patterns are supported in OpenWFE, jBPM or Enhydra Shark.

3.8 Trigger patterns

The trigger patterns distinguish between transient and persistent triggers. Transient Triggers (WCP-23) are lost if not acted upon immediately (which implies that they need to be waited for at a particular place in a process) and can appear both in a safe execution environment (where only one process instance at a time can be waiting for a trigger to arise) and an unsafe environment. The specific feature for the second case is that only one instance can be activated by a trigger, which implies that triggers are distinct and can be uniquely matched to the awaiting process instances. Persistent Triggers (WCP-24) are triggers retained by the process until they can be acted upon. They are either buffered until the task which they target is reached or they are able to trigger tasks that are not dependent of the completion of any preceding tasks.

Every task, i.e., <participant> definition in OpenWFE can be timed out (see Listing 39, where an instance of task1 will be skipped if not performed within 10 minutes). The default setting for timeout is one week. It is defined through the engine variable __time_out__ and can be changed either by directly modifying its value in the configuration file or by (re)setting it from a process. To prevent a task to be timed out, its *timeout* attribute has to be set to “no”. While the timeout construct is convenient for defining back-up schemes that prevent process instances from stalling midway through their execution, they are more of a management tool than a real trigger.

With the <sleep> construct, the execution of a task can be delayed (see Listing 40, which is taken from the OpenWFE specification [22]. The time for the delay is either relative (sleep for) or absolute (sleep until). The expiration of the time interval defined for the <sleep> construct triggers the continuation of the flow. When the time is relative, every trigger (time expiration) is mapped to a specific process instance. When the time is absolute (and defined up through global engine variables) its expiration may result in several process instances being triggered. The definition of a <sleep> construct at a particular point in the process, ensures that the trigger is expected and acted upon immediately when received. Hence through this construct OpenWFE provides support for transient triggers.

Listing 38 The when construct in OpenWFE - not working

```
<concurrence>
   <when>
      <equals variable-value="/triggerA"
         other-value="13" />
      <participant ref="role-alpha"
         description="taskA"/>
   </when>
   <sequence>
      <participant ref="role-bravo"
         description="task1"/>
      <set variable="/triggerA" value="13"/>
      <participant ref="role-bravo"
         description="task2"/>
   </sequence>
</concurrence>
```

Listing 39 Timout in OpenWFE

```
<sequence>
   <participant ref="role-bravo"
      description="task1" timeout="10m"/>
   <participant ref="role-bravo"
      description="task2"/>
</sequence>
```

Listing 40 The sleep construct in OpenWFE

```
<participant ref="role-bravo" />
<sleep for="10m" />
<participant ref="role-alpha" />
<set variable="date"
   value="$c:tadd('$c:now()','3d')" />
<sleep until="$date" />
<participant ref="role-bravo" />
```

No support for persistent triggers was found in OpenWFE. The persistent trigger behaviour cannot be simulated through the use of a <when> construct within a <concurrence> flow, as exemplified in Listing 38. The <when> construct operates as an asynchronous <if>, which repetitively evaluates the specified condition (with the frequency defined in the configuration files) and when true executes the specified action. The surrounding <concurrence> expression is used to keep the <when> expression active during the execution of the remaining part of the process (which in this example is defined as a sequence). In the example an internal process variable “/triggerA” is monitored through the when-condition and a change in its value effectively acts as a trigger. The time interval to the next when-condition re-evaluation effectively causes a delay in the delivery of the trigger. For capturing external triggers, engine variables (e.g. “//triggerA”) could be used, however this is potentially problematic in unsafe scenarios as no distinction can be made between various process instances. However, the most serious difficulty with this solution is that, as a consequence of continually re-evaluating the when-construct, the process never ends.

Transient triggers are supported in jBPM through the notion of State. When a process instance reaches a state on its execution path, it waits for a signal in order to leave the state and continue execution. The concept of Timer is also present in jPDL. However, currently timers do not work in the engine when XML process definitions have been deployed to the engine and instances of them were initiated from the web console interface [17] (which are the scenarios we are interested in). No support for persistent triggers was observed in jBPM.

Data Visibility	1	2	3	Data Interaction-External (cont.)	1	2	3
1. Task Data	+/–	–	+/–	21. Env. to Case–Push-Oriented	–	–	–
2. Block Data	–	+	+	22. Case to Env. –Pull-Oriented	–	–	–
3. Scope Data	–	+/–	–	23. Workflow to Env.– Push-Oriented	–	–	–
4. Multiple Instance Data	–	+	+	24. Env. to Process – Pull-Oriented	–	–	–
5. Case Data	+	+	+	25. Env. to Process – Push-Oriented	–	–	–
6. Folder Data	–	–	–	26. Process to Env.– Pull-Oriented	–	–	–
7. Global Data	–	+	–	Data Transfer			
8. Environment Data	+/–	+	+/–	27. by Value – Incoming	–	–	+/–
Data Interaction-Internal				28. by Value – Outgoing	–	–	+/–
9. Task to Task	+	+	+	29. Copy In/Copy Out	+	+	+
10. Block Task to Subprocess Decomp.	–	+	+	30. by Reference – Unlocked	–	–	–
11. Subprocess Decomp. to Block Task	–	+	+	31. by Reference – Locked	–	+	–
12. to Multiple Instance Task	–	–	–	32. Data Transformation – Input	+	+	+
13. from Multiple Instance Task	–	–	–	33. Data Transformation – Output	+	+	+
14. Case to Case	+/–	+/–	+/–	Data-based Routing			
Data Interaction-External				34. Task Precondition – Data Exist.	–	+	–
15. Task to Env. – Push-Oriented	+/–	+	+	35. Task Precondition – Data Value	–	+	–
16. Env. to Task – Pull-Oriented	+/–	+	+	36. Task Postcondition – Data Exist.	–	–	–
17. Env. to Task – Push-Oriented	–	–	–	37. Task Postcondition – Data Val.	–	–	+/–
18. Task to Env. – Pull-Oriented	–	–	–	38. Event-based Task Trigger	–	–	–
19. Case to Env. – Push-Oriented	–	–	–	39. Data-based Task Trigger	–	–	–
20. Env. to Case – Pull-Oriented	–	–	–	40. Data-based Routing	+/–	+/–	+

Table 2. Support for the Data Patterns in **1**–JBoss jBPM 3.1.4, **2**–OpenWFE 1.7.3, and **3**–Enhydra Shark 2.0

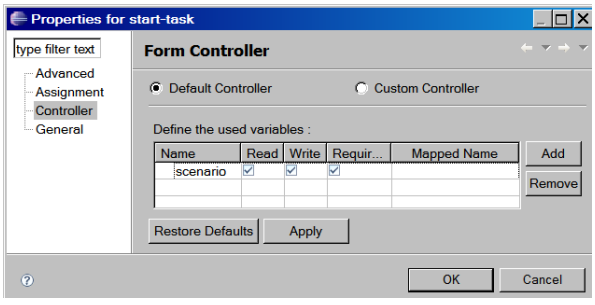
Enhydra Shark indicates that there is some support for transient triggers through the specification of a Deadline on an activity. However, the deadline functionality did not work in our tests. Another typical example of the use of transient triggers is activities denoting email reception. A ToolAgent in Enhydra defines the interface to a data source or an application in the operational environment and a MailToolAgent is dedicated for implementing the connectivity needed for sending and receiving e-mail. However, in the evaluated version of the tool, the configuration needed for invoking the available MailToolAgent did not work. Enhydra Shark does not support persistent triggers. In order to receive a trigger (such as a deadline expiration or a mail reception), a process needs to be executing a specific task that can handle that trigger. This corresponds to the notion of transient trigger.

4 Data Patterns Evaluation

The data patterns describe the way in which data is represented and utilized in the context of a workflow system. In this section, the results of a patterns based evaluation of the data perspective for jBPM, OpenWFE, and Enhydra Shark are presented. Table 2 provides a summary of these evaluations using the same three-point scale as was previously used for the control-flow evaluations.

The data patterns divide into five main groups: *data visibility patterns* characterizing the various ways in which data elements can be defined and utilised; *internal* and *external data interaction patterns* dealing with the various ways in which data elements can be passed between components within a process instance and also with the operating environment; *data transfer patterns* focusing on the way in which data elements are actually transferred between one process element and another; and *data routing patterns* capturing the various ways in which data elements can interact with other perspectives and influence the overall execution of the process. The evaluation results for each of these groups of patterns are discussed in turn in the following sections. However, first the approach that each of these three systems takes to the representation of data within workflow processes is introduced.

Figure 23 Graphical interface of controllers in jBPM



Listing 41 Variables in jBPM

```
<task name="start-task" swimlane="ernie">
  <controller>
    <variable name="scenario"
      access="read,write,required">
  </controller>
</task>
```

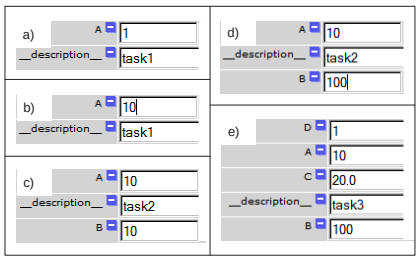
In **jBPM**, by default variables have a scope that encompasses the entire workflow process. They are created as part of the definition of an individual task via the *controller* construct. The intended function of a controller is to define the mapping between task and process variables. Currently only a one-to-one mapping is supported. (If a more complex mapping needs to be represented, a specific TaskControllerHandler capturing it would need to be implemented first.) Controllers not only deal with the mapping of process and task data, but they are also the only place where variables (both task and processes) can be defined. Figure 23 shows the graphical interface of a controller (and the definition of the variable "scenario" from the start task in Figure 5). Listing 41 shows the corresponding XML representation. Mapped Name(s) define the task variable name, where it is different from the global variable naming (defined under Name). If specified, Mapped Name will be the name displayed to the user at runtime. The settings Read, Write and Required define whether the value in the task variable will be read from the corresponding process variable (and if the process variable has not yet been created, it is created), written back to it (if so, any previous value is overwritten) and whether the user input of a value for the variable is compulsory[13].

In **OpenWFE** data handling is realized through variables and fields. *Fields* hold data values, which are populated and/or accessed by end users, while *variables* hold data values which are used for internal processing and are not visible to or accessible by end users. Data can be transferred between fields and variables at any time during process execution other than during the execution of tasks. After being defined, a field is visible in all consecutive tasks unless its visibility is limited through the use of filters. In contrast, defined, variables have a lifespan corresponding to that of the process instance. In the documentation [20], a distinction is made between local (subprocess) variables (which are denoted by the name of the variable) e.g. *varname*, process variables, e.g. */varname* (which denoted by a slash preceding the name), and engine variables, e.g. *//varname* (which are denoted by two slashes preceding the name). Operations on data elements (other than user input) can not be done inside a (atomic) task, instead data elements must be manipulated when they are passed between the tasks.

Listing 42 Variables and Fields in OpenWFE

```
1  <sequence>
2     <set field="A" type="integer" value="1" />
3     <set variable="/x" value="empty" />
4     <set variable="//x" value="${f:A}" />
5     <participant description="task1" ref="role-alpha" />
6     <set variable="/x" value="${f:A}" />
7     <set field="B" type="integer" value="${x}" />
8     <participant description="task2" ref="role-bravo" />
9     <set field="C"
10       value="${call:sum('${x}' '${f:A}')}" />
11    <set field="D" value="${//x}" />
12    <participant description="task3" ref="role-alpha" />
13 </sequence>
```

Figure 24 Fields interfaces in OpenWFE



Listing 42 illustrates variable and field definitions in a process and Figures 24a-e show how they appear in the user interface when the process is run. In Listing 42, field A and the variables /x and //x are defined prior to the commencement of "task1". Field A is assigned the value '1' as a default, /x is assigned the value "empty" and //x is assigned the value of field A, (which is '1'). Figure 24a provides a screenshot of part of the user interface which indicates the values of data elements at the commencement of "task1". Note that only field A (and not the variables) appear in this and that this field is writable by the user. Figure 24b shows a change in the value in A to 10 as a result of a change made by an end user. When 'task1' completes execution, the process continues to 'task2' but before it commences, the value of variable /x is assigned the value of field 'A', which is now 10 (see line 6 in the listing) and field 'B' is created with its default value set to that of variable x (see line 7). Note that variable x has not been defined. In practice, OpenWFE does not differentiate between subprocess and process scope (the moment a subprocess variable /x is defined, a corresponding process variable x is also created) which means that x is assigned the value of /x, hence field B is assigned the value '10'. At the commencement of 'task2' the values for both field A and B are as shown in Figure 24c. Both fields are writable by the end user and in Figure 24d the value of field 'B' is changed to 100 during the execution of the task. After the completion of 'task2', field C is defined (line 9 in the listing) and its initial value derived from the values of variable x and field A (line 10), which illustrates simple data manipulation. Finally, field D is defined and set to the value of the variable //x (line 11 in the listing), which is displayed at the commencement of 'task3' (see Figure 24e).

Enhydra Shark uses variables to store data relevant to a process. A variable is declared to be one of several types, e.g., Array or Basic. The default variable type is Basic with sub-types such as String or Integer (see Figure 25, left).

[13] During our test, variables with the setting Required (such as the variable scenario) were found to accept empty values, i.e. the Required setting did not work as expected.

New types of variables can be formed by either combining pre-defined types or restricting pre-defined ones. Also at design time variables may be given default values.

In Enhydra Shark a package aggregates one or several workflow processes. A variable may be defined at the process level or at the package level (see Figure 25, right) using the Datafield tag. A datafield positioned at the process level defines a variable to be accessible by all instances of the particular process, whereas a datafield positioned at the package level defines a variable to be accessible by all process instances in the package. A consequence of this is that a variable defined at the package level may be used and reused by several processes. A package level variable, however, is not a global variable in the usual sense. When instantiating a process a copy of the variable is made accessible for the process instance. This copy resides entirely within the process instance and is not visible outside it. A package level variable cannot be used as a means for communicating between process instances.

At design time, the execution of an activity can be defined to involve reading or writing the value of a variable. For example, a transition condition from one activity to the next may be dependent on a specific value of a variable. A user executing the activity at run time will then be confronted with an input box to change the value and enable the transition. Variables can also be defined for manipulation during an automatic procedure. For instance, a variable may be incremented to keep count of iterations in a loop. The administration console allows the inspection of variables at any time during process execution.

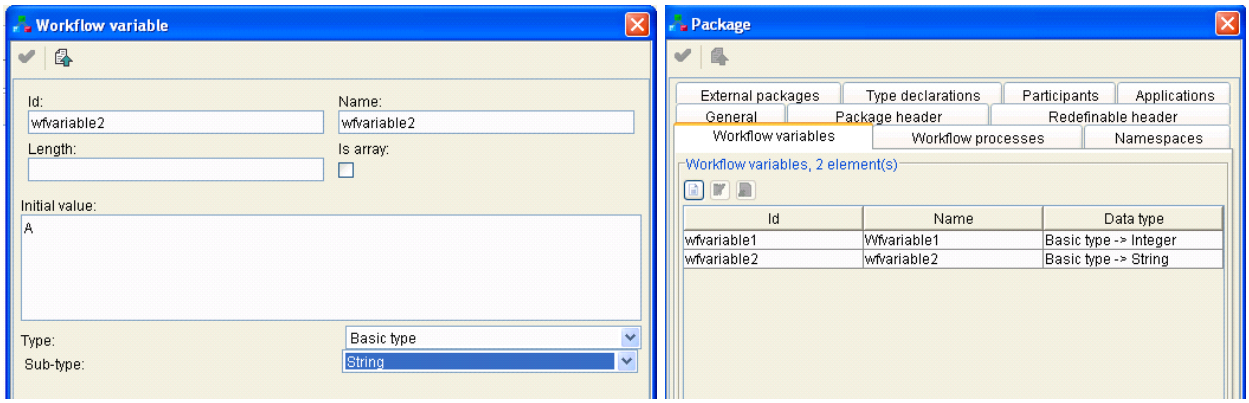


Figure 25 Variables in Enhydra Shark

4.1 Data visibility patterns

As previously indicated, in jBPM the most common approach to characterising data elements is via process variables hence, the Case Data pattern (WDP-5) is directly supported. However, owing to the fact that task variables are mapped to process variables on a one-one basis and the inherent difficulty of defining task variables that do not strictly correspond to existing process variables, the Task Data pattern (WDP-1) is not considered to be fully supported.

jBPM provides the notion of Super State which is intended to be used for grouping different nodes into logical units and hence serves as a mechanism for hierarchical structuring of processes. A corresponding symbol is also present in the graphical editor. However, it is not currently possible to graphically define the components which make up a Super State. In addition, the notion of Process State is introduced as a means of facilitating the invocation of subprocesses which are described in terms of distinct process fragments to the main process definition. However the use of this construct is problematic and leads to deployment failure. Hence, block structures are not implemented and there is no support for the Block Data pattern (WDP-2). The task-node construct can accommodate several tasks and can potentially be considered as a partial implementation of the scope concept. However, no data elements can be defined for a task-node, thus the Scope Data pattern (WDP-3) is not supported.

As indicated in Section 3, jPDL lacks any notion of multiple instance tasks. If such tasks are required as part of a process model, a corresponding implementation of their behaviour would need to be completed first. jBPM provides the ability to embed such an extension within a node-type Node construct however the need for additional programmatic effort to achieve this outcome means that the Multiple Instance Data pattern (WDP-4) is not considered to be directly supported. Nor are any of the other visibility patterns supported.

In OpenWFE, all data elements are globally accessible throughout a process instance and, once defined, all fields are visible (unless restricted by the use of filters) to all subsequent tasks in a process instance. Hence OpenWFE directly supports the Case Data pattern (WDP-5) and consequently the Task Data pattern (WDP-1) is not supported. The presence of engine variables that can be accessed at any point in any process (e.g. *//varname*) confirms that Global

Data (WDP-7) is supported. When a subprocess definition resides in the same file as the main process, a subprocess variable (e.g. */varname*) is also visible and changeable from the main process. However local binding can be enforced through the use of a dot as prefix (i.e. *.varname*), hence OpenWFE also supports the Block Data pattern (WDP-2).

Filters can be used to specify read and write access restrictions on certain fields. Listing 43 shows an example of the definition of a filter called 'student_view' and its use with a specific task (i.e. task 'register'). The setting type="closed" (on line 12) implies that any field not explicitly specified within the filter will not be visible when the filter is applied. The setting of attributes *add* and *delete* to "false" implies that additional fields can not be added or erased at runtime (which is not the case for the default setting). Four attributes are defined before the application of the filter to task 'register' and during the execution of the task only the fields 'name' and 'address' will be displayed (in addition to the __description__ attribute of the task) while the attributes 'result1' and 'result2' will be filtered out. In particular the field 'end_result', which is included in the filter, is not displayed during the execution of 'register', as it has not been defined before the task.

Listing 43 Filters in OpenWFE

```
1 <sequence>
2     <set field="name" type="string" value=""/>
3     <set field="address" type="string" value=""/>
4     <set field="result1" type="integer" value="0"/>
5     <set field="result2" type="integer" value="0"/>
6     <participant description="register"
7      ref="role-student" filter="student_view"/>
8     ...
9 <sequence>
10
11 <filter-definition name="student_view"
12         type="closed" add="false" remove="false">
13     <field regex="name" permissions="rw"/>
14     <field regex="address" permissions="rw"/>
15     <field regex="end_result" permissions="r"/>
16     <field regex="__description__" permissions="r"/>
17 </filter-definition>
```

Listing 44 A Web-Service Operation defined as a Participant in OpenWFE

```
1 <participant name="ws-weather">
2     ...
3     <param>
4         <param-name>wsdl</param-name>
5         <param-value>http://www.webservicex.net/
6         globalweather.asmx?WSDL</param-value>
7     </param>
8     <param>
9         <param-name>operationName</param-name>
10        <param-value>GetWeather</param-value>
11    </param>
12    <param>
13        <param-name>signature</param-name>
14        <param-value>city, country</param-value>
15    </param>
16    <param>
17        <param-name>return_to_field</param-name>
18        <param-value>weather</param-value>
19    </param>
20 </participant>
```

The use of filters for limiting access to fields qualifies as partial support for the Scope Data pattern (WDP-3) as the filter overwrites the default privileges for the data element and as such restricts rather than extends the access rights associated with it. As the number of data elements associated with a process instance tends to increase during its execution, the use of filters for restricting data flow can become quite complex.

OpenWFE supports the Multiple Instance Data pattern (WDP-4) as is clear from Listing 33 in the previous section. At runtime each task instance that is created has its own value for the variable 'user'. The Environment Data pattern (WDP-8) is also supported and external data in the operating environment can be accessed via web-services or external applications. In OpenWFE these are specified as participants in the participant-map configuration file. Listing 44 provides an example where the operation GetWeather from the Web-Service Global Weather is defined as an OpenWFE participant[14], which means that it can, just like any other participant in the system, be invoked (i.e. have work distributed to it) from anywhere in a process specification. The *return_to_field* parameter specifies the location to which the environment data is transferred.

Enhydra Shark too offers process scoped variables, hence it supports the Case Data visibility pattern (WDP-5). It does not directly support Task Data (WDP-1), as data elements are defined at the package level or the process level. However, as it is possible to in a task define Javascript procedures, which can contain local data (such as loop counters), the support for Task Data is considered to be partial. (Similarly, through the Node construct in jBPM, task data can be defined and utilized. As this is achieved programmatically, the support for Task Data in jBPM is rated as partial.) Furthermore, Enhydra Shark supports Block Data visibility (WDP-2), through the notion of subprocess variables which are local to the corresponding subprocesses. In Enhydra Shark it is possible to define block activities. A block activity is an activity that encapsulates a set of activities into an embedded subflow. Data in a block activity can be shared with the containing process through parameter passing. That is, parameters of the block are mapped to variables in the calling process.

Through the notion of subprocess, Enhydra Shark supports shared implementation of tasks, hence it supports Multiple Instances Data (WDP-4). Global Data (WDP-7) is not supported. While variables are available at the package level (a package contains one or several workflows), these variables contain Case Data, i.e., their data values are not shared between several cases. Hence the benefit of these variables is mainly to reduce the amount of variable definitions within a package.

[14] This example is a part of the default OpenWFE 1.7.3 installation.

Enhydra Shark supports Environment Data (WDP-8) through the use of Tool Agents. A Tool Agent provides an interface to a data source or an application in the operational environment. One tool agent, e.g. JavaScriptToolAgent (providing connectivity to the local Javascript interpreter), is available with the default installation in the evaluated product; hence this pattern is considered to be supported. Because coding in Java is needed, the support is considered as partial.

4.2 Data interaction patterns - internal

In jBPM, variables have process scope, hence the Data Interaction - Task to Task pattern (WDP-9) is supported through the global shared data approach. As there is no support for multiple instance tasks in jBPM and the fact that all attempts to implement task decomposition failed, there is no support for data interaction patterns involving multiple instances, nor for data interaction to and from subworkflows. Achieving a similar effect through programmatic extensions is not considered to constitute partial support according to the evaluation criteria for these patterns. Support for the Data Interaction – Case-to-Case pattern (WDP-14) in jBPM is rated as partial as it can only be indirectly achieved through programmatic interaction with an external database, for example, by associating an action with a transition (as suggested in Section 9.5 of the documentation [12]).

OpenWFE provides a broader range of facilities for internal data interaction. Through the use of global variables, the Data Interaction – Task to Task pattern (WDP-9) is supported. Global variables are also utilized for data passing to and from blocks/subprocesses (WDP-10, WDP-11). Subprocesses are defined either in the same file as the main process from which they are invoked (see line 11 and lines 16-21 in Listing 45) or in a separate file (thus allowing different processes to invoke them as a part of their execution), see Listing 46 and line 12 in Listing 45. Unless limited through the use of filters, all fields (e.g. field 'a' and field 'next') become automatically visible to every subprocess, both internal and external, that is invoked after their definition. Furthermore, a variable (e.g. variable 'x') is accessible directly (i.e. both readable and writable) from any subprocess by using its name preceded by a slash (see lines 6-7 in Listing 46). When the subprocess is defined in the same file as the main process, the slash preceding the name of the variable can be omitted (as is done on line 18 in Listing 45). In addition, OpenWFE also supports data interaction with a subprocess through the use of parameters. For an example of this, see lines 11 and 19 in Listing 45 where the value of field 'next' is passed to the subprocess 'internalB' through the parameter 'pr'.

Listing 45 Internal Data Interaction in OpenWFE

```
1  <process-definition name="dp10-11" revision="1">
2    <description>Data transfer-main</description>
3    <sequence>
4      <set field="a"  type="integer"  value=""/>
5      <set variable="x"  type="integer"  value="2"/>
6      <set field="next" type="string" value="role-?"/>
7      <participant description="task A" ref="role-alpha"/>
8      <if>
9         <lesser-than value="${call:mul('${f:a}' '${v:x}')}"
10             other-value="10" />
11        <subprocess ref="internalB" pr="${f:next}"/>
12        <subprocess ref="http://localhost:7079/sub.xml"/>
13     </if>
14     <participant description="task C" ref="role-alpha"/>
15   </sequence>
16   <process-definition name="internalB">
17     <sequence>
18       <set variable="x" value="${call:mul('2' '${v:x}')}"/>
19       <participant ref="${pr}" description="task B"/>
20     </sequence>
21   </process-definition>
22 </process-definition>
```

Listing 46 External subprocess in OpenWFE - file sub.xml in the workflow-definitions directory

```
1  <process-definition name="externalB"
2                      revision="1">
3    <description>dp10-11 - subprocess
4    </description>
5    <sequence>
6       <set variable="/x" value="${call:
7             mul('4' '${v:/x}')}" />
8       <participant ref="role-student"
9             description="task B-ext"/>
10   </sequence>
11 </process-definition>
```

Specific data elements can be passed to the instances of a multiple instance task, i.e. WDP-12 (as illustrated earlier in listings 32-34 in Section 3 where the concurrent-iterator construct was used to initiate multiple concurrent instances of a task), but aggregation of the data produced by these instances (WDP-13) is not supported. Instead, the data elements of one of the instances of the multiple instance task are taken and used in subsequent tasks. The attributes *merge* and *merge-type* can be used to define how the data from the different instances should be merged. Possible values for the attribute *merge* are 'first', 'last', 'highest' and 'lowest' and for the attribute *merge-type*, the values 'mix' and 'override' are defined. The attribute setting merge='last' and merge-type='mix' (as shown on lines 13 and 14 in Listing 34) implies that the data values for the last of the instances to complete will be taken and if any data values are missing they will be filled using the values from the previously completed instances.

The Data Interaction – Case to Case pattern (WDP-14) is only partially supported in OpenWFE as data is passed between cases using global variables (see the definition of the workflow variable //x on line 4 in Listing 42 for an example of this). This is considered to constitute partial support for the pattern as it results in the more general

communication of a data element from one case to *all* other cases, and not to one specific case as is the intention of the pattern.

Enhydra Shark supports the Data Interaction - Task to Task pattern through global (process scoped) shared variables. It also supports data interaction between a Block Task and its Subprocess Decomposition. Global shared variables are used for data interaction to and from a Block Activity which is embedded in the main process, while parameter passing is used for communication to and from subflows that are defined outside of the main process. Data interaction to and from multiple instance tasks is not supported as the concept of task data is not supported. The Data Interaction - Case to Case pattern is partially supported through interaction with an external database. Similar to jBPM and OpenWFE, the more general behaviour of passing data from one case to all other cases is captured but not data passing from one case to another specific case.

4.3 Data interaction patterns - external

OpenWFE supports both the Data Interaction – Task to Environment – Pull and Data Interaction – Environment to Task – Push patterns (i.e. WDP-15 and WDP-16). Listing 48 shows an example of a process capable of sending email to name@company.com, with the content of the field="__subject__" in it (i.e. the Data Interaction – Task to Environment – Push-oriented pattern (WDP-15)). The mail is sent by a task called "send mail" executed by participant "notif-alpha", the latter being an artificial participant configured in the participant-map file as such (an excerpt of which is shown in Listing 47). Listing 49 demonstrates invocation of a web-service (and consequently demonstrates support for the Data Interaction – Task to Environment – Pull Oriented pattern (WDP-16)). To be able to invoke an external service, a corresponding participant needs to be defined in the participant-map configuration file. Listing 44 shows how the participant *ws-weather* corresponding to the operation *GetWeather* service *globalweather* is defined. The values retrieved from the fields *city* and *country* (as defined in lines 2 and 3 in Listing 49) during the execution of task "request-service" are sent to the *ws-weather* participant (line 6 Listing 49) when invoking the *GetWeather* operation and the answer received is assigned to the field *weather* (as defined in line 18 in Listing 44) during the execution of the task "display result" (lines 7 and 8 in Listing 49). Although OpenWFE does not provide support for task data (and instead supports case data), the data interaction is initiated at the task level rather than the case level, and hence is bound to a specific place/task in the process. Consequently the patterns Data Interaction – Case to Environment – Push and Pull oriented (WDP-19 and WDP-20) are not considered to be supported and neither are the patterns Data Interaction – Process to Environment – Push-oriented and Pull-oriented (WDP-23 and WDP-24).

Although it is possible to use an RDBMS for persistence and OpenWFE is distributed with code for MySQL, PostgreSQL, MS-SQL and DB2 connectivity, it is not considered that this functionality directly facilitates support for data transfer between a process and external sources. It is really intended as a means of achieving data persistence in a WFMS, hence such a DBMS is considered to be an integral part of the WFMS (rather than of the environment).

Listing 47 WDP-15 in OpenWFE - excerpt from a participant configuration

```
1 <participant name="notif-alpha"
2 class="[..].MailNotifiedParticipant">
3     <!-- email notification parameter -->
4     <param>
5         <param-name>recipient-field</param-name>
6         <param-value>to</param-value>
7     </param>
8     <!-- parameters with their default values -->
9     <param>
10        <param-name>smtp-server</param-name>
11        <param-value>mail.company.com</param-value>
12    </param>
13    <param>
14        <param-name>smtp-port</param-name>
15        <param-value>25</param-value>
16    </param>
17    ...
```

Listing 48 WDP-15 in OpenWFE - excerpt from a process definition

```
1 <sequence>
2     <set field="to" value="name@company.com" />
3     <set field="__subject__" value="test" />
4     <participant description="send mail"
5         ref="notif-alpha" />
6 </sequence>
```

Listing 49 WDP-16 in OpenWFE - excerpt from a process definition

```
1 <sequence>
2     <set field="city" value="Stockholm" />
3     <set field="country" value="Sweden" />
4     <participant ref="role-bravo"
5         description="request service" />
6     <participant ref="ws-weather"  />
7     <participant ref="role-bravo"
8         description="display result" />
9 </sequence>
```

Similar to OpenWFE, external data interaction in jBPM is realised at task (and transition) level, rather than at the process level. The Data Interaction – Task to Environment – Push pattern (WDP-15) is most easily realised through Actions, which are defined on transitions or the entrance or the exit of nodes. In [6] pp 134-140, the connection of a jBPM task to an external database and the writing of data to that database is exemplified, by actually coding an ActionHandler Java class which enables the required data transfer. In a similar way, through the use of Nodes which can stop and wait for the receipt of data which they are retrieving from external sources, the Data Interaction – Environment to Task – Pull pattern (WDP-16) can also be implemented. Hence patterns WDP-15 and WDP-16 are considered to be supported. However, due to the extent of Java coding required in achieving the required data

interaction, the support for these patterns is considered to be partial. None of the other patterns in this category are supported by jBPM.

From this group of patterns, Enhydra Shark only offers support for the Task to Environment - Push and Environment to Task - Pull-Oriented patterns. A ToolAgent supplies an Enhydra Shark's interface towards different software. For instance SOAPToolAgent can be used for invoking web-services to which data can be sent and from which data can be received. The invocation of a tool agent is specified through ExtendedAttributes.

4.4 Data transfer patterns

Data is transferred in jBPM using a Copy in/Copy out strategy (i.e. WDP-29). When dealing with concurrent tasks, this implies that at creation every work item receives a copy of the data values in the global process variables. Any updates to these values by a concurrent work item remains invisible to that work item. At completion the values for each work item are copied back (if so specified) to the global process variables, overwriting the existing values for these variables. This means that the last work item to complete will be the last one to copy its values back to the process variables. To avoid potential problems with lost updates, the process designer needs to be careful when granting write access to variables for tasks that run in parallel and update global data.

Scripts can be used to manipulate data elements before a task is started and after it has completed, i.e. at the initiation and completion of a task node (see lines 2-3 and 12-13 in Listing 50)[15]. The event types "node-enter" and "node-leave" indicate when the script will be run. A script consists of an expression, describing what needs to be done, and a set of variables, specifying the input data to the script (if a variable definition is omitted, all process variables will be loaded into the script upon evaluation). When a task-node contains a single task, the node-enter and node-leave scripts provide direct support for the two data transformation patterns WDP-32 and WDP-33.

Listing 50 Data transformation in jBPM

```
1  <task-node name="task1">
2    <event type="node-enter">
3      <script>
4        <expression> z = x + " " + y;
5        <variable name="A" access="read" mapped-name="x">
6        <variable name="B" access="read" mapped-name="y">
7        <variable name="C" access="write" mapped-name="z">
8      </script>
9    </event>
10   <task name="task1"> ...
11   <transition name="" to="task2">
12   <event type="node-leave">
13     <script>
14       <expression> c = c +" "+ c;
15       <variable name="C" access="read,write"
16                 mapped-name="c">
17     </script>
18   </event>
19 </task-node>
```

In terms of the Data Transfer patterns, OpenWFE provides direct support for the Data Transfer by Reference With Lock pattern (WDP-31). As all data elements are global for a process, each of them resides in a location accessible by all process components. Concurrency restrictions are implied such that when a task is started by a resource, the corresponding work item (and associated data elements) is locked and the remaining resources with execution privileges for the task receive temporary read-only access to it. When dealing with concurrent tasks or multiple instance tasks, OpenWFE utilizes a Copy in/Copy out strategy (i.e. WDP-29). At creation, every work item receives a copy of the data values in the global variables. Any updates to these values by a given work item remain invisible outside that work item. At completion the values associated with the work item are copied back in accordance with the strategy defined in the synchronization construct, which has the potential to lead to lost update problems. Data transfer to and from multiple instances tasks is treated similarly (as discussed in the second paragraph of Section 4.2). To avoid the potential for lost updates, filters can be used on concurrent tasks to prevent simultaneous access to data elements. However, filters can not be used to solve this issue for multiple instances tasks.

Data transformations on input and output data (patterns WDP-32 and WDP-33 respectively) are supported. As previously mentioned, data transformations can be performed at any time between two tasks in a process. Examples of such transformations are illustrated by lines 6-7 in Listing 46 (where the value of the process variable /x is multiplied by four before the commencement of task "task B-ext") and lines 9 and 10 in Listing 42 (where the value of the field C is calculated based on input from the preceding task "task2").

Similar to jBPM and OpenWFE, Enhydra Shark supports transformations on input and output data (patterns WDP-32 and WDP-33) and Copy In/Copy Out (WDP-29) patterns. However the copy out operation is not applied when concurrent work items are operating on the same data, which may lead to data being lost. When invoking subprocesses or block activities in Enhydra Shark, it is also possible to transfer data using parameters, hence there is support for the Data Transfer by Value – Incoming and Data Transfer by Value – Outgoing patterns (WDP-27 and WDP-28). The

[15] jPDL also supports scripts at the creation, start, assignment and end of a task. However, the graphical interface does not provide support for the definition of such scripts. Moreover, the task scripts we defined in the XML files specifying the processes did not execute at runtime.

support is considered as partial because these strategies are not applicable for data transfer between any activities, but only applicable for data transfer to and from block activities and subprocesses.

4.5 Data-based routing

There is no support for the Task Precondition - Data Existence (WDP-34) or Data Value (WDP-35) patterns in jBPM or Enhydra Shark, as there is no means to specify preconditions. OpenWFE supports both these patterns. The Task Precondition - Data Value pattern is illustrated through the if-statement on lines 8-13 in Listing 45, where if the sum of values in field 'a' and variable 'x' is less than ten, the internal subprocess is invoked otherwise an external subprocess is invoked. The Task Precondition – Data Existence pattern is supported in a similar way, except that the condition associated with the <if> statement contains an expression <defined variable-value="variable-name"> testing whether the variable *variable-name* has been defined or not.

Enhydra Shark also partially supports the Task Postcondition – Data Value pattern (WDP-37). The support is partial because only the data types (but not their values) are checked. Furthermore, Enhydra Shark does not support the Task Postcondition – Data Existence pattern (WDP-36). Conditions like "varName != null" and "varName == null" can be specified, but while the first of these conditions always evaluates to true, the second one always evaluates to false because if a value is not entered, Enhydra Shark puts an empty string in the variable (varName in this example). If the variable varName is an integer, an error message stating "Incorrect type!" is displayed.

In jBPM, the Task Postcondition – Data Existence pattern (WDP-36) is intended to be supported through the attribute Required for variables. However, it was found that not providing a value for a variable which was marked as required did not prevent the user from registering the task as completed[16]. Hence there is no support for postconditions checking whether certain parameters have received a value, nor is there any support for the Task Postcondition – Data Value pattern (WDP-37). No support for the two postcondition patterns was observed in OpenWFE.

The Event-based Task Trigger pattern (WDP-38) is an extension of the Transient and Persistent Trigger patterns (CFP-23 and CFP-24), describing triggers in which data (and not just signals) is transferred. Furthermore, the Data-based Task Trigger pattern (WDP-39) focuses on the situation where the trigger is not an external event (e.g. mail arrival) but the change of an internal data condition.

Support for Data-based Task Triggers was not observed in OpenWFE. The pattern is currently not supported in jBPM as stated in jBPM [15], nor does Enhydra Shark support it. None of the offerings supports Event-based Task Triggers. In jBPM the State node can only be used for receiving signals as triggers where there is no data attached to the signals. In the evaluated version of Enhydra Shark, the MailToolAgent was tested for receiving mails as triggers, but did not work.

The support for the Data-based Routing pattern (WDP40) is rated as partial in both jBPM and OpenWFE as there is not full support for the multi-choice variant of the pattern. In contrast, Enhydra Shark fully supports the Data-based Routing pattern as it supports both the exclusive and multiple choice constructs.

5 Resource Patterns Evaluation

The resource perspective focuses on the manner in which work is distributed between the resources in an organization and managed through to completion. In workflow management systems this translates to the way that tasks and task instances (i.e. work items) are distributed among the participants (i.e. the users) associated in a process. Figure 26a (reprinted from [24]) provides a general state transition diagram for the lifecycle of a work item. The prefixes S and R in the names of the transitions are abbreviations for *System* and *Resource* and indicate the initiator of a specific transition. A work item is first created by the system, which means that the preconditions for the enablement of the associated task have been met. At this point in the process, an instance of the task (also known as a work item) is created. This work item is either offered to a single resource or to a group of resources, or directly allocated to a specific resource. When it is offered, the offer is non-binding, i.e. a resource can choose to accept or reject it, and the work item remains on offer until it is finally selected by one of the resources to which it is offered. Once selected by a resource, the work item is allocated to that resource on a binding basis (i.e. the resource is expected to complete the work item at some point in the future). Subsequent to being allocated, a work item can be executed, (i.e. started and completed) at a time chosen by the resource. During execution the work can be suspended and resumed. The execution may also lead to unsuccessful competition and result in the work item ending up in a failed state. Some workflow engines allow a work item to be started immediately after it has been offered, thus omitting the allocation step (and hence expediting work item commencement).

[16] The discussion on whether it is a bug in jBPM or a failure in JSF which assigns an empty string to variables when no value has been entered for them is ongoing. For details see [16].

Creation Patterns	1	2	3	Pull Patterns, continuation	1	2	3
1. Direct Allocation	+	–	+	24. System-Determ. Work List Mng	–	–	–
2. Role-Based Allocation	–	+	+	25. Resource-Determ. Work List Mng	–	–	–
3. Deferred Allocation	+	+	+	26. Selection Autonomy	+	+	+
4. Authorization	–	–	–	Detour Patterns			
5. Separation of Duties	–	–	–	27. Delegation	–	–	–
6. Case Handling	–	–	–	28. Escalation	–	+	–
7. Retain Familiar	+	–	–	29. Deallocation	–	+	+
8. Capability-based Allocation	–	–	–	30. Stateful Reallocation	–	+	–
9. History-based Allocation	–	–	–	31. Stateless Reallocation	–	–	–
10. Organizational Allocation	–	–	–	32. Suspension/Resumption	+	–	–
11. Automatic Execution	+	+	+	33. Skip	–	–	–
Push Patterns				34. Redo	–	+/–	–
12. Distribution by Offer-Single Rsr.	–	–	+	35. Pre-Do	–	–	–
13. Distribution by Offer-Multiple Rsr.	–	+	+	Auto-start Patterns			
14. Distribution by Alloc.-Single Rsr.	+	–	–	36. Commencement on Creation	–	–	–
15. Random Allocation	–	–	–	37. Commencement on Allocation	–	–	–
16. Round Robin Allocation	–	–	–	38. Piled Execution	–	–	–
17. Shortest Queue	–	–	–	39. Chained Execution	–	–	–
18. Early Distribution	–	–	–	Visibility Patterns			
19. Distribution on Enablement	+	+	+	40. Config. Unallocated WI Visibility	–	+/–	–
20. Late Distribution	–	–	–	41. Config. Allocated WI Visibility	–	+/–	–
Pull Patterns				Multiple Resource Patterns			
21. Resource-Init. Allocation	–	–	–	42. Simultaneous Execution	–	–	–
22. Resource-Init. Exec. - Allocated WI	+	–	–	43. Additional Resources	–	–	–
23.Resource-Init. Exec. - Offered WI	–	+	+				

Table 3. Support for the Resource Patterns in **1**–JBoss jBPM 3.1.4, **2**–OpenWFE 1.7.3, and **3**–Enhydra Shark 2.0

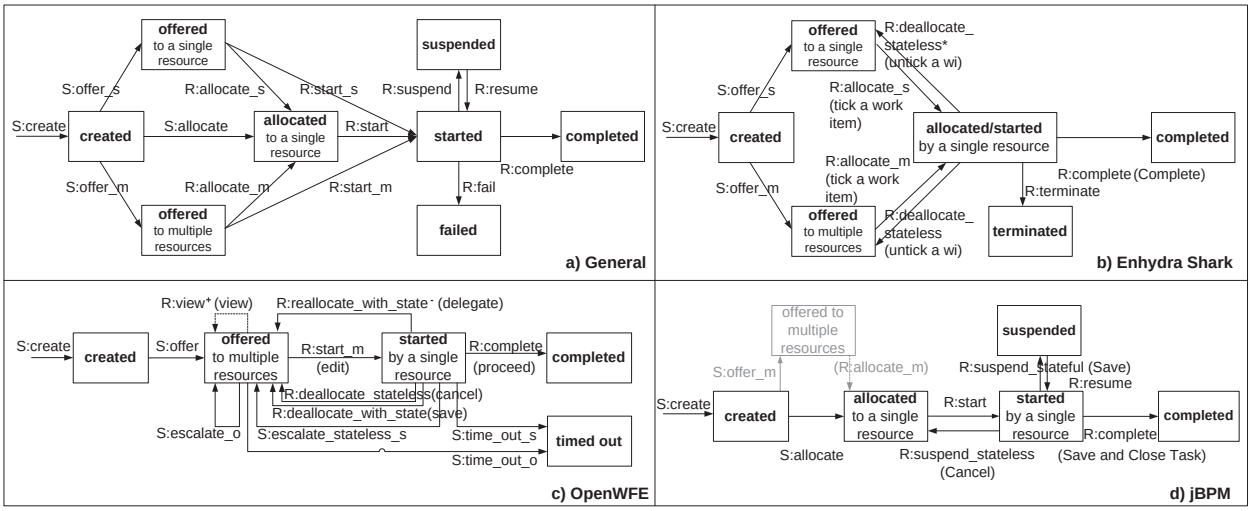


Figure 26 The lifecycle of a work item (the transition names given in parentheses are the corresponding commands in the user interfaces for these tools)

The majority of the resource patterns are grouped with reference to the work item lifecycle and the patterns are divided into the following categories: *creation patterns*, *push patterns*, *pull patterns*, *detour patterns*, *auto-start patterns*, *visibility patterns*, and *multiple resources patterns*. Before discussing how the three offerings support the patterns in these groups, it is worth mentioning that the notion of *offered* is missing in jBPM, the notion of *allocated* is missing in OpenWFE and the *allocated* and *started* states are merged in Enhydra Shark. Hence their lifecycle diagrams, presented in Figures 26b-d, deviates from the one shown in Figures 26a. These diagrams will be further explained throughout this section. The results of the evaluation are summarised in Table 3.

5.1 Creation patterns

The creation patterns describe the various ways in which a work item is handled after creation and prior to it being started. They are specified at design time and rely on the existence of an associated organizational model where resources can uniquely be identified and their roles and relationships within the organization are captured. In the work item lifecycle diagram this category of patterns correspond to the S:create transition. The major patterns belonging to this category are: *Direct Allocation* - which captures the ability to assign a task to a specific resource; *Role-based*

Allocation - which depicts the ability to assign a task to a specific role (such that any resource playing this role can potentially execute it); *Deferred Allocation* - which allows the decision regarding how a task will be distributed to one or more resources to be deferred to runtime; *Retain Familiar* - which captures the ability to allocate a work item to the same resource that completed a preceding work item in the same case; and *Automatic Execution* - which describes the ability of a task to be executed without it needing to be distributed to a human resource. Descriptions of the remaining creation patterns can be found in [24].

In jBPM, when a work item is created, it is directly assigned to a specific resource and included in the work list for that resource (which constitutes support for the Direct Allocation pattern). An example of a work list for user 'ernie' can be found in Figure 3. The use of assignment expressions in jPDL for the purposes of defining how tasks are distributed to users is included in Listing 51.

Listing 51 WRP-1 Direct Allocation in jBPM

```
<task-node name="concurrence A">
   <task name="task A1">
      <assignment expression="user(ernie)">
   </task>
   <task name="task A2">
      <assignment expression="user(bert)">
   </task>
</task-node>
```

Listing 52 WRP-2 Role-based Allocation in jBPM

```
<task-node name="concurrence A">
   <task name="task A1">
      <assignment expression="group(programmers)">
   </task>
   <task name="task A2">
      <assignment expression="group(testers)">
   </task>
</task-node>
```

Listing 53 WRP-2 Role-based Allocation in OpenWFE

```
<concurrence>
   <participant description="task A1"
      ref="role-programmer">
   <participant description="task A2"
      ref="role-tester">
</concurrence>
```

Listing 54 WRP-1 Direct Allocation in OpenWFE - not supported

```
<sequence>
   <participant description="task A1"
      ref="user-ernie">
   <participant description="task A2"
      ref="user-bert">
</sequence>
```

In OpenWFE, when a work item is created, it is offered to a role (thus corresponding to the Role-based Allocation pattern). For an example of how this is configured, see Listing 53. The work items distributed to a role are added to the worklist for a store based on the specification in the worklist-configuration file (refer back to Listing 2 for further details). These work items can then be executed by any user with read and write permissions for that store (see Listing 3 for an example of how this is specified in the passwd configuration file).

The Direct Allocation pattern is not supported in OpenWFE, i.e. it is not possible to assign work items directly to users as proposed in Listing 54. The flow halts in this listing at runtime because the users allocated to the tasks within it are not recognized and the corresponding work items do not get distributed to them. Direct allocation can be achieved by configuring only one user to have access to a given store, but as this solution is really a special case of role-based allocation it is not considered to constitute support for the Direct Allocation pattern. Furthermore, as there is no means of defining an organizational structure in OpenWFE, i.e. there is no support for specifying relationships between users and roles, or any more general forms of organizational groupings and hierarchy, the Organisational Distribution pattern is also not considered to be supported.

Theoretically, jBPM also supports the ability to offer a work item to a group of resources. jPDL contains the primitives "user", "group", "membership" and "role" and provides the ability to allocate tasks to whole groups, thus potentially supporting the Role-based Allocation pattern (see Listing 52). Furthermore there are language primitives for distributing work to different roles played by the participants belonging to a specific group, and these could conceivably be used for supporting the Organizational Distribution pattern (see Listing 55). However, while the necessary language primitives are envisaged, the web-console's current implementation (distributed with the jBPM 3.1.4 package[17]) does not provide any support for them. In particular, any selection by a resource of a work item offered to a group (of which it is a member) is not supported (i.e. the system does not provide support for a transition R:allocate_m in Figure 26). For the moment, an expression such as the one in Listing 52 leads to a stalled work item which is only visible in the database managing the persistence, but which does not appear in any of the available worklists. For this reason, Role-based Allocation is not considered to be supported. Interestingly, the resource allocation in Listing 55 works when there is only one resource playing the specified role. In this case the work item is put in its worklist. If more than one resource plays the same role however, the flow stalls. Hence the Organizational Distribution pattern is also considered not to be supported.

[17] The console in the latest release of the tool utilized during these evaluations, i.e. jbpm-jpdl 3.2.1, did not provide support for resource management. Any resource was able to start and complete any task independently of the process definitions in jPDL. For that reason we did not evaluate it further, preferring to stay with the older but more stable version for the other evaluation activities.

Listing 55 WRP-10 Organizational Distribution in jPDL

```
<task-node name="concurrence A">
   <task name="task A1">
      <assignment expression="group(employees)
         -->role(programmer)"> </task>
   <task name="task A2">
      <assignment expression="group(employees)
         -->role(tester)"> </task>
</task-node>
```

Listing 56 CFP-14 MI and WRP-3 Deferred Allocation in OpenWFE

```
<sequence>
  <set field="iteration_list"
     type="StringMapAttribute"
     value="users?: alpha,bravo.."/>
  <participant ref="role-alpha"
     description="distribute tests to subjects"/>
  <concurrent-iterator
     on-field-value="iteration_list"
     to-field="user">
     <participant ref="role-${f:user}"
        description="run test"/>
  </concurrent-iterator>
  <participant ref="role-alpha"
    description="summarize results"/>
</sequence>
```

Listing 57 WRP-3 Deferred Allocation and WRP-7 Retain Familiar in jBPM

```
<start-state name="start">
   <task name="start task">
      <assignment expression="user(ernie)">
      <controller>
         <variable name="empl"
         access="read,write,required">
      </controller>
   </task>
   <transition name="" to="task1">
</start-state>
<task-node name="task1">
   <task name="task1">
      <assignment actor-id="empl">
   </task>
   <transition name="" to="task2">
</task-node>
<task-node name="task2">
   <task name="task2">
      <assignment expression="previous">
   </task>
   ...
</task-node>
```

Enhydra Shark has very limited support for the creation patterns. It supports Direct, Role-based, and Deferred Allocation. A Participant in the workflow model can be mapped both to a specific User and to a Group of users defined in the process engine. The screenshot in Figure 12 shows how the participant "Seller" (from the process model in Figure 11) is mapped to the User "petia" and the participant "buyer" is mapped to the Group of users called "admin". Deferred allocation is implemented by using variables. A Performer variable can be specified for an activity and its value set during runtime.

Deferred Allocation is also supported by jBPM and OpenWFE. Similarly to Shark, it is implemented using variables which receive their values at runtime before the commencement of the relevant task. The assignment of "task1" on line 13 in Listing 57 shows how variable "empl", populated by user ernie during the execution of "start-task" is used for Deferred Allocation in jBPM. For OpenWFE a similar example is shown in Listing 33 (for convenience this listing is reprinted as Listing 56), where the solution for one of the Multiple Instance patterns was presented.

The Retain Familiar pattern is only supported in jBPM. The resource assignment for "task2" on line 19 in Listing 57 where the expression attribute is set to "previous" shows how this is implemented. Using this assignment expression for every task in a process would in practice result in all tasks in a case being executed by the resource who started the case. Note that this is considered as repeated use of the Retain Familiar pattern, rather than support for the Case Handling pattern, as every task needs to have an individual assignment expression (as opposed to a whole case, when created, being allocated to one resource). Consequently this solution is not considered to provide support for the Case Handling pattern.

The Automatic Execution pattern, which allows tasks to be executed by applications and is common to most workflow systems is supported by all three offerings.

5.2 Push patterns

The push patterns describe the different ways in which a work item is offered or allocated to resources by the system. These include indirect offerings where a work item is advertised on a shared work list, as well as direct offering and allocation schemes where a work item is distributed to the work list of a specific resource. There are three groups of patterns belonging to this category. The first group captures the way in which the distribution is done. The three patterns in this group are: *Distribution by Offer-Single Resource*, *Distribution by Offer-Multiple Resources* and *Allocation-Simple Resource* which correspond directly to the transitions S:offer_s, S:offer_m and S:allocate in Figure 26a. The second group captures the different means by which a resource is singled out from a set of possible candidates and allocated a work item. The patterns in this group are *Random Allocation*, *Round Robin Allocation* and *Shortest Queue* allocation. The third group focuses on the timing of the allocation process and the availability of a work item. It distinguishes three patterns: *Early Distribution*, *Distribution on Enablement* and *Late Distribution*.

Characteristic for all three offerings is that when a work item is created, it is immediately distributed, thus illustrating support for the Distribution on Enablement pattern. Furthermore, jBPM currently supports Distribution by

Allocation to a Single Resource pattern[18], OpenWFE supports Distribution by Offer to Multiple Resources pattern and Enhydra Shark supports Distribution by Offer to Single Resources and Distribution by Offer to Multiple Resources patterns. None of the allocation patterns (random, round robin or shortest queue) are supported, as these patterns are only relevant when a system selects a resource from a number of possible candidates and this does not occur in any of the offerings.

5.3 Pull patterns

The pull patterns capture situations where resources that have been offered work items commit themselves to execute them at some future time. The situation where a resource simultaneously selects and commences the execution of an offered work item is also included in this category.

There are six pull patterns: *Resource-Initiated Allocation* which captures the situation where a resource commits to executing an offered work item at some future time (illustrated by transitions R:allocated_s and R:allocated_m in Figure 26a); *Resource-Initiated Execution - Allocated Work Item* where a resource commences a work item already allocated to it (depicted by the transition R:start in the lifecycle diagram); *Resource-Initiated Execution - Offered Work Item* where a resource simultaneously commits to executing and starts an offered work item (illustrated by the transitions R:start_s and R:start_m in Figure 26a); *System/Resource-Determined Work Queue Content* which describe the ability of the system/resource to control the content and the ordering of the work items in a work list; and *Selection Autonomy* which describes the ability of a resource to select the execution order for the work items in their work list.

In jBPM, a work item that is allocated to a resource can be started by the resource selecting the work item from their work list. This indicates support for the Resource-Initiated Execution – Allocated Work Item pattern. If several work items are available in a work list, the order of their execution is determined by the resource which also indicates support for the Selection Autonomy pattern. None of the other pull patterns are supported in jBPM.

In OpenWFE, a work item offered to a shared group list (e.g. as in Figure 9) can be simultaneously selected and started by a resource in the group selecting it from their work list (i.e. selecting "edit" for that work item). Thus the Resource-Initiated Execution – Offered Work Item pattern is directly supported. The order in which work items are executed is determined by individual resources based on the contents of their work list hence the Selection Autonomy pattern is also supported. None of the other pull patterns are supported in OpenWFE.

In Enhydra Shark, a work item offered to a user appears in his/her work list (see Figure 13); a work item offered to a group, appeares in the work lists of every member of the group. Then the work item can be simultaneously selected and started by a resource (in the group). When selected/started by a resource, a work item that was offered to a group, disappears from the work lists of the remaining group members. Thus the Resource-Initiated Execution – Offered Work Item pattern is supported. If several work items are available in a work list, the order of their execution is determined by the resource, which indicates support for the Selection Autonomy pattern.

5.4 Detour patterns

Detour patterns refer to situations where the normal process by which work items are distributed to resources and managed through to completion is interrupted, resulting in a varied sequence of states and resource assignments for such work items. The interruption can be instigated by a resource or by the system. One such pattern captured in Figure 26a is *Suspension/Resumption*, which denotes the possibility for a resource to temporarily suspend the execution of a work item and to resume it at a later time. To avoid cluttering the diagram with atypical state transitions, the remainder of the detour patterns are not shown graphically. They are: *Delegation* which denotes the situation where a resource allocates a work item from their work list to another resource (in Figure 26a this would be captured by a transition from the "allocated" state back to itself); *Escalation* which describes the situation where the system proactively attempts to progress a work item that has stalled, by offering or allocating it to another resource or group of resources (in Figure 26a this would be represented by a number of possible transitions, each one going from the "started", "allocated", or "offered" states back to one of these states); *Deallocation* which denotes the situation where a resource relinquishes a work item which is currently allocated to them and make it available for (re-)distribution to other resources; *Reallocation* which captures the situation where a resource (re-)allocates a work item that they have started working on to another resource. In contrast to the Delegation and Deallocation patterns, Reallocation addresses work items that have been started. Also, in contrast to the Escalation pattern, Reallocation is initiated by a resource. A reallocation is either *Stateful* (i.e. it preserves the state of a work item which is captured by the data associated with it) or *Stateless*. In the work item lifecycle, these patterns correspond to transitions from the "started"

[18] In the latest tested release i.e. 3.2.1, jBPM seems to be preparing support for Distribution by Offer-Multiple Resources and the topic is frequently discussed on their forum, however such support is not yet available.

state back to the "allocated" state; *Skip* describes the scenario where a resource omits the execution of a work item assigned to them. This corresponds to a transaction from the "allocated" state to the "completed" state; *Redo* which provides the possibility for a resource to execute a work item that has already been completed, thus introducing a backward transition from the "completed" to the "started" state; and *Pre-Do* which denotes the ability of a resource to execute a work item ahead of the current execution point in a case (a possibility that does not have a corresponding representation in the workitem lifecycle diagram).

Once a work item has been started in jBPM, it can be completed through the "Save and Close Task" option (see Figure 5), or it can also be suspended by selecting one of "Save" and "Cancel" options for stateful or stateless suspension (hence demonstrating support for the Suspend/Resumption pattern, which is also shown in the lifecycle in Figure 26d).

In Enhydra Shark a user can "release" a task she selected and initiated (by un-checking it in the work list). If it is a work item that initially was offered to a group of users this will result in re-offering the work item to the group. Hence the Deallocation pattern is supported, though the semantics deviates slightly from this for the pattern. As the state "allocated" is merged with the state "started" (see Figure 26b), a work item needs to be started in order to be deallocated. This implies that a distinction can be made between stateful and stateless deallocations. The deallocation supported in Enhydra Shark is stateless.

In OpenWFE, a started work item is completed by selecting the "proceed" option from the task execution window (see Figure 7). Once it is started, a work item can be reallocated to another group of resources which is achieved by selecting the "delegate" option. As the task must have been started in order to select the "delegate" option, the behaviour does not correspond to the Delegation pattern, but instead to the Reallocation pattern. This form of reallocation is stateful hence the Stateful Reallocation pattern is directly supported. Reallocation of a work item can only be initiated by users who possess the "delegate" right (defined in the passwd configuration file, recall Listing 3) on the store for which they are executing the work item in question. This is indicated in Figure 26c through the use of the suffix '–' for the R:allocated_with_state transition. Delegation is not supported as a work item can only be forwarded to another resource once it has started. In addition, a started work item can also be deallocated and selecting the "cancel" option results in a stateless deallocation with the work item being sent back to the group work list. Selecting the "save" option results in a stateful deallocation. Note that this is distinct from the Suspension/Resumption pattern as the interrupted work item does not remain in the work list of the resource who started executing it, but is sent back to the group work list and could potentially be continued by any resource that is a member of the same group. Note also that the semantics deviate slightly from those for the Deallocation pattern. As the state "allocated" is missing in OpenWFE, a work item needs to be started in order to be deallocated. This implies that a distinction can be made between stateful and stateless deallocations.

Enhydra Shark does not support Stateful or Stateless Reallocation. The Reassign button in the Work List Console is currently not enabled (see Figure 13). It does not support Suspension/Resumption of a work item. In the evaluated version of the tool a "Suspension" and a "Resumption" button for a work item are displayed but not enabled for use. In Enhydra Shark a whole case can be suspended and resumed. In such a situation all work items available for the case are suspended (and resumed). They still appear in the work list(s), but they are not possible to allocate/start (when trying to do that a wrong error message is displayed, i.e. "the work item is allocated to another resource").

OpenWFE supports the Escalation pattern, although the fact that the change of resource for the work item is initiated by the system means that it is not supported from the user interface shown in the screenshots in Figures 9 and 7. Escalation is captured via the "timeout" attribute through which the time duration for a task is limited to a specified value. For instance, in Listing 58 the duration of "task1" is limited to 30 seconds. After a task is timed out (which may occur both when the task is waiting to be started or after it has actually been started) the flow either continues to the next task (which in the lifecycle diagram is shown by the S:time_out_s and S:time_out_o transitions) or the timed-out work item is (re-)distributed to another group work list (captured in the work item lifecycle diagram through the S:escallate_stateless_s and S:escallate_o transitions). The new resource allocation, as is the case in the listing, needs to be explicitly defined through an if-then statement specifying the identity of the new resource assigned to the task. As indicated by the name S:escalate_stateless_s the escalation is stateless.

Enhydra Shark does not support Escalation. The notion of deadline is present for limiting the duration of an activity, but during runtime the deadline did not expire. The example provided with the installation package did not work either.

Listing 58 WRP-28 Escallation in OpenWFE

```
<sequence>
   <set field="__subject__" value="task1" />
   <set field="first name" value="name" />
   <participant ref="role-bravo"
      description="task1" timeout="30s"/>
   <if>
      <equals field-value="__timed_out__"
         other-value="true" />
      <participant ref="role-alpha"
         description="task1" />
   </if>
   <participant ref="role-charly"
      description="task2"/>
</sequence>
```

The existence of the <cursor> construct may at a first glance give the impression that the behaviour of the Skip pattern can be captured in OpenWFE. Tasks that are included within the <cursor> and </cursor> tags, may at runtime be given the value "skip" due to the __cursor_command__ field appearing in their user interface. However, this does not influence the execution of the current work item, which still needs to be completed by pressing the "proceed" button. As with any other completed work item, this work item is recorded in the execution log as having successfully completed with all the changes in the data made during its execution being preserved. In the __cursor_command__ field, the value 'skip 2' can be given in order to skip the subsequent (next-following) task. However even this action does not provide support for the Skip pattern, because the work item for the subsequent task does not get created, which is a prerequisite for the Skip pattern.

The Redo pattern is supported in OpenWFE through the concept of the <cursor>. An important issue associated with this pattern is that if a user goes back a number of steps and redoes a task, then they are forced to also redo any task that may be influenced by the redone task, i.e. the act of redoing a task should not break the integrity of the overall process model. This can be achieved by only allowing the "back" and "rewind" operations within a cursor (and disallowing the "skip", "break" and "cancel" operations). However, when a process is rewound a number of steps, the task to be redone is offered not only to the resource who initiated the rewind, but to all members of the shared worklist, and it can therefore potentially be executed by anyone with sufficient execution rights. Furthermore, the rewind command, operates not at a task level but at the instruction level. This means that, for example, if there is a data-manipulation instruction between two tasks, it will be included as part of the rewind steps. This necessitates very detailed knowledge of the process model by the end user when rewinding a process instance to a desired task, and requires that they are not only aware of the number of tasks that they will need to rewind, but also cognisant of any data manipulation operations and transfers defined in between tasks when determining the steps to be rewound. As a consequence of these peculiarities, support for the Redo pattern in OpenWFE is only considered to be partial.

The Pre-Do pattern is not supported in any of the tools. The pattern implies that a task can be performed in isolation before the thread of control has reached the task. This behaviour is not possible in any of the three tools.

5.5 Auto-start patterns

Auto-start patterns relate to situations where the execution of a work item is triggered by specific events in the lifecycle of the work item or the related process definition. Such events may include the creation or allocation of the work item, completion of another instance of the same work item or a work item that immediately precedes the one in question. There are four auto-start patterns: *Commencement on Allocation* (represented by the S:start transition in the work item lifecycle from the "allocated" to the "started" state) which describes the situation where a work item is started at the same time that it is allocated to a specific resource; *Commencement on Creation* which is similar to the Commencement on Allocation pattern in that it describes a transition triggered by the system, but in this case the commencement (and allocation) of the work item occurs directly after it is created. The rationale for this behaviour is that it expedites the throughput of work items in a process instance by cutting out some of the intermediate states (albeit with some resultant loss of flexibility in the deployment of the process). The remaining two patterns in this category capture the initiation of a work item based on the completion of another, related, work item. *Piled Execution* denotes the ability of the system to initiate the next work item for a given task (possibly in a different case) immediately after a resource has completed another instance of the same task. *Chained Execution* describes the ability of the system to automatically start the next work item in a case for a resource once they have completed the preceding work item. It differs from the Retain Familiar pattern in that the new work item is not only allocated to the same resource, but it is also started for them.

None of the auto-start patterns are supported in jBPM, OpenWFE or Enhydra Shark as all of them relate to situations where the system starts the execution of a work item 'on behalf' of a resource. In the evaluated offerings, only a resource can initiate the execution of a work item assigned to them. In Enhydra Shark, "Automatic" can be selected as a Start Mode for an activity (the default start mode is "Manual"). However, at runtime the work items of activities with this "Automatic" setting are not started automatically, but need to be started manually by selecting them from the corresponding work list.

5.6 Visibility and multiple resources patterns

Unlike the previous groups of resource patterns, the *visibility patterns* define process characteristics that are not directly related to the work item lifecycle. Instead, they describe the openness of a system's operation to scrutiny by process participants. There are two visibility patterns: *Configurable Unallocated* and *Allocated Work Item Visibility* which describe the ability to configure the visibility of unallocated and allocated work items to resources involved in a process.

OpenWFE provides limited support for both these patterns. When a user is defined (see Listing 3), the privileges they have to access different stores are also specified. Giving "read" privileges to a user for a store implies that they will be able to see both unallocated work items and also started (i.e. allocated) work items in that store. The fact that only users with read privileges can see the workitems in a store is depicted with the suffix '+' in the R:view transition in Figure 26c[19]. This configuration cannot be performed by the user, but is performed by a workflow administrator. It is interesting to note that it is not possible to configure the visibility so that only started or only unallocated items are visible. Hence the degree of support for the visibility patterns in OpenWFE is considered to be partial. There is no support for these patterns in jBPM or Enhydra Shark.

Finally, the *multiple resource patterns* describe situations where there is not a one-to-one correspondence between executing work items and resources (which is assumed to hold for the other resource patterns). There are two patterns in this group: *Simultaneous Execution* which describes the ability of a resource to execute several work items concurrently; and *Additional Resources* which describes the situation where more than one resource is assigned to the same work item. Neither of these patterns are supported by jBPM, OpenWFE or Enhydra Shark.

6 Conclusions

This paper sought to investigate the state-of-the-art in open source workflow management software through a patterns-based analysis of three representative systems. The use of a patterns-based evaluation framework provides an independent means of assessing the ability of each offering to support a wide range of business process constructs that commonly occur in practice.

Overall, one can conclude that the range of constructs supported by the three systems is somewhat limited, although OpenWFE tends to offer a considerably broader range of features than jBPM and Enhydra Shark. There are several potential areas for improvement and further development in all the offerings. From a control-flow standpoint, jBPM and Enhydra Shark support a relatively limited set of control-flow operators offering little support for the patterns outside the basic control-flow category. OpenWFE provide a better range of facilities for task concurrency but has limited support for the OR-join construct and its Webclient does not provide any runtime user cancellation operators either at task or process level.

From a data perspective, all three offerings support a limited range of data element bindings and rely heavily on case-level data elements. Having said that, the data passing strategies employed in the three systems whilst simplistic are reasonably effective and include consideration of important issues such as inline data manipulation whilst data elements are being passed. There are however limited capabilities for handling external data interaction without programmatic extensions. It is noticeable that jBPM relies heavily on the use of Java for addressing data-related issues and thus its overall level of direct support for the data patterns is relatively low. OpenWFE provides a more comprehensive support in this perspective and supports a wider (but still limited) range of features. In Enhydra Shark external data communication is meant to be supported through predefined Tool Agents, however some of these (e.g. MailToolAgent) did not work in the evaluated open-source version of the offering.

A notable shortcoming in all three offerings is the minimalistic support for the data perspective to influence other aspects of workflow operation, esp. the control-flow perspective e.g. no (or limited) postconditions, trigger support and limited data-based routing support. Another concern are the shortcomings when dealing with data manipulation activities occurring in parallel. When parallel work items are operating on the same data: jBPM copies back the corresponding values in the order of work items' completion (overwriting the values of earlier completed instances); OpenWFE copies back the corresponding values according to the specified strategy i.e. First, Last, etc (overwriting data when the Last strategy is applied and ignoring new data when the First strategy is applied); and Enhydra Shark does not copy back the variable values (hence also losing data).

In the resource perspective, only simple notions of work distribution are supported and typically only one paradigm exists for work item routing in each offering. There is no support for any form of work distribution based on organizational criteria, resource capabilities, or execution history. All three offerings provide relatively simple facilities for

[19] Note also that this transition is slightly different than the other transitions in the lifecycle diagram namely, it does not imply any real change between the states that are its source and target. To show this difference a dotted arrow is used in the diagram.

work item management; there is no or limited ability to configure work lists at the resource or system level, no notion of concurrent work item execution and no facilities for optimizing work item throughput (e.g. automated work item commencement, chained execution). One area where OpenWFE demonstrates noticeably better facilities is in terms of the range of detour patterns (e.g. deallocation, reallocation) that it supports.

It is interesting to compare the state-of-the-art in open source workflow systems with the state-of-the-art in proprietary systems. For the latter we refer to the evaluations presented in [27]. The results of these evaluations are summarised on [9]. A particular area of strength for the proprietary systems is that of resource patterns support, an area in which they appear to be significantly more mature than the open source offerings examined.

While the patterns framework can be applied successfully to analyse and compare the suitability of offerings for business process management, additional aspects need to be considered when assessing the maturity of an offering before deploying it in production. Examples of such aspects are quality of documentation, development maturity (i.e. the degree to which an offering satisfies the promises made by its vendor), administration facilities, degree of openness, ease of installation and configuration, and availability of support. With respect to these aspects our experiences indicate that the open source workflow environments examined leave considerable room for improvement, and below some of these are discussed.

OpenWFE lacks explanatory documentation for its graphical notation, the user management tool UMAN, and the command line administration tool called Control. This resulted in process models being directly specified in XML, user management achieved through updates in the configuration files, and administration difficulties when faced with cancellation of work items and process instances. All of this requires a deep understanding of operational aspects of the tool.

jBPM's claim of fully supporting the original set of 20 control-flow patterns is rather overstated, as for many patterns the system still relies on additional Java coding for their implementation. Furthermore, process decomposition defined through the jPDL language primitives Process State and Super State in the graphical editor did not work. These parts of the language were also poorly documented hence decomposition defined directly in the XML code was not possible. Finally data entries defined as Required at design time could be left empty at runtime.

Enhydra Shark's vendor Together, while providing an open-source workflow engine, also provides a closed-source Together Workflow Server and some desirable functionality (e.g. administration and user functionality) is only present in the latter version. In this regard it is also worthwhile to mention that a rather negative experience with Enhydra Shark's Process Engine was the frequent appearance of a nagging pop-up window containing vendor information (which appeared approximately every fifth minute and which blocked work for ten seconds).

As regards administration support, all three tools fell a bit short of the expectations. An example is the support for user management. In jBPM, SQL statements are required to define users. In Enhydra Shark special settings in one of the configuration files are necessary for user groups defined through the TWS Admin client and, as mentioned, OpenWFE lack of documentation complicated user management.

Though support was not evaluated, it is worth mentioning that the OpenWFE help forum provided fast and generally useful assistance. In terms of interface design of the Process Engine and the Worklist handler, Enhydra Shark stood out. The installation of both OpenWFE and Enhydra Shark was straightforward. jBPM provides a convenient introductory web-tutorial.

Overall one can conclude that the open source systems are geared more towards developers than towards business/software analysts. If one is intimately familiar with Java, jBPM may be a good choice, while if this is not the case, choosing jBPM is less advisable. Similarly, while OpenWFE has a powerful language for workflow specifications in terms of its support for the workflow patterns, we postulate that this language will be difficult to understand by non-programmers. On the other hand Endydra Shark's weak support for the workflow patterns may require complicated work arounds for capturing nontrivial business scenarios.

Acknowledgement

We would like to thank John Mettraux for prompt and helpful responses on the OpenWFE help forum and Saša Bojanic for constructive and valuable feedback on Enhydra Shark's results from the draft version of this paper.

Disclaimer

We, the authors and the associated institutions, assume no legal liability or responsibility for the accuracy and completeness of any product-specific information contained in this paper. However, we have made all possible efforts to make sure that the results presented are, to the best of our knowledge, up-to-date and correct.

References

1. W.M.P. van der Aalst, M. Dumas, and A.H.M ter Hofstede. Web Service Composition Languages: Old Wine in new Bottles. In *In Proc. of 29th EUROMICRO Conf., Track on Software Process and Product Improvement*, pages 298–307, 2003.
2. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
3. Alfred Madl et.al., Together Teamlösungen EDV-Dienstleistungen GmbH. Together Workflow Editor - User Guide. `http://www.together.at/together/zzznocms/twe/twedoc/twe.html`, 2007. Last accessed 05 Dec 07.
4. B. Kiepuszewski and A.H.M. ter Hofstede and C. Bussler. On Structured Workflow Modelling. In B. Wangler and L. Bergman, editors, *Proc. of the 12th Int. Conf. on Advanced Information Systems Engineering (CAiSE00)*, volume 1789 of *LNCS*, pages 431–445. Springer, 2000.
5. Workflow Management Coalition. Workflow Reference Model. Available at `www.wfmc.org/standards/referencemodel.htm`. Last accessed 27 Sep 07.
6. M. Cumberlidge. *Business Process Mamagement with JBoss jBPM: a Practical Guide for Business Analysts*. Packt Publishing, Birmingham, UK, 2007.
7. Enhydra.org. Open Source Java XPDL Workflow. `http://www.enhydra.org/workflow/shark/index.html`. Last accessed 20 Nov 07.
8. Paul Harmon. Exploring BPMS with Free or Open Source Products. BPTrends, Available at `www.bptrends.com/publicationfiles/advisor200707311%2Epdf`, July 31 2007. Last accessed 27 Sep 07.
9. Workflow Patterns Initiative. Workflow Patterns - homepage. Available at `www.workflowpatterns.com`. Last accessed 27 Sep 07.
10. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. Thomson Computer Press, London, UK, 1996.
11. Java-source.net. Open Source Workflow Engines in Java. Available at `java-source.net/open-source/workflow-engines`. Last accessed 27 Sep 07.
12. JBoss. JBoss jBPM 3.1 Workflow and BPM made practical. Available at `docs.jboss.com/jbpm/v3.1/userguide/en/html/taskmanagement.html`. Last accessed 27 Sep 07.
13. JBoss. JBoss jBPM 3.1 Workflow and BPM made practical, Chapater 9.6 Superstates. Available at `docs.jboss.com/jbpm/v3.1/userguide/en/html/processmodelling.html`. Last accessed 18 Dec 07.
14. JBoss. JBoss jBPM website. Available at `www.jboss.com/products/jbpm`. Last accessed 27 Sep 07.
15. JBoss jBPM forum. Data trigger task end. Available at `www.jboss.com/index.html?module=bb&op=viewtopic&t=102283`. Last accessed 27 Sep 07.
16. JBoss jBPM forum. Required variables. Available at `www.jboss.com/index.html?module=bb&op=viewtopic&t=118219`. Last accessed 27 Sep 07.
17. JBoss jBPM forum. Timer problem. Available at `www.jboss.com/index.html?module=bb&op=viewtopic&t=119301`. Last accessed 27 Sep 07.
18. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003. Available via `http://www.workflowpatterns.com/documentation/documents/phd_bartek.pdf`.
19. Manageability. Open Source Workflow Engines Written in Java. Available at `www.manageability.org/blog/stuff/workflow_in_java`. Last accessed 27 Sep 07.
20. John Mettraux. The OpenWFE Manual: Open source WorkFlow Environment. Available at `www.openwfe.org/manual/index.html`. Last accessed 27 Sep 07.
21. OpenWFE. OpenWFE website. Available at `www.openwfe.org`. Last accessed 27 Sep 07.
22. OpenWFE. OpenWFEru, Time Expressions, sleep. Available at `http://www.openwfe.org/manual/ch06s08.html#expression_sleep`. Last accessed 27 Sep 07.
23. OpenWFE. OpenWFEru, Workflow Patterns, Structural Patterns. Available at `http://openwferu.rubyforge.org/patterns.html#pat_c`. Last accessed 27 Sep 07.
24. N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In O. Pastor and J. Falcao é Cunha, editors, *Proc. of the 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520 of *LNCS*, pages 216–232. Springer, 2005.
25. N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM Center Report BPM-06-22, BPMcenter.org, 2006.
26. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In L. Delcambre et al, editor, *Proc. of the 24th Int. Conf. on Conceptual Modeling (ER 2005)*, volume 3716 of *LNCS*, pages 353–368. Springer, 2005.
27. N.C. Russell. *Foundations of Process-Aware Information Systems*. PhD Thesis, Queensland University of Technology, June 2007. Available at `http://www.yawl-system.com/theory/publications.php`.
28. WfMC. Workflow Process Definition Interface - XML Process Definition Language, Version 1.0. Accessed Dec 2007 from `http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf`, 2002.