COVER SHEET

Event-based Coordination of Process-oriented Composite Applications

Marlon Dumas[1], Tore Fjellheim[1], Stephen Milliner[1], and Julien Vayssière[2]

[1] Queensland University of Technology, Australia
(t.fjellheim,s.milliner,m.dumas)@qut.edu.au
[2] SAP Research Centre, Brisbane, Australia
julien.vayssiere@sap.com

Abstract. A process-oriented composite application aggregates functionality from a number of other applications and coordinates these applications according to a process model. Traditional approaches to develop process-oriented composite application rely on statically defined process models that are deployed into a process management engine. This approach has the advantage that application designers and users can comprehend the dependencies between the applications involved in the composition by referring to the process model. A major disadvantage however is that once deployed the behaviour of every execution of the composite application is expected to abide by its process model until this model is changed and re-deployed. This makes it difficult to enrich the application with even minor features, to plug-in new applications into the composition, or to hot-fix the composite application to meet special circumstances or demands (e.g. to personalise the application). This paper describes a technique for translating a process-oriented application into an event-based application which is more amenable to such runtime adaptation. The process-based and event-based views of the application can then co-exist and be synchronised offline if the changes become permanent and it is found desirable to reflect them in the process model.
Keywords: flexible process execution, activity diagram, event-based coordination, coordination middleware, object space.

1 Introduction

Process-oriented composite applications aggregate functionality from a number of other applications by specifying interconnections between these applications through a process model. This model determines how the underlying applications should be orchestrated, most notably their dependencies in terms of flow of control and data. Mainstream infrastructures for developing and executing process-oriented composite application include workflow management systems and process management modules embedded within Enterprise Application Integration (EAI) solutions. Predefined process models can be deployed into the runtime environments associated to these infrastructures for execution.

A major advantage of using a process-oriented approach for composite application development is that it provides an easy-to-comprehend and global view of the dependencies between the underlying applications. However, in existing process-oriented systems these dependencies have to be completely specified before deployment [1]. In certain environments, such as mobile computing, changes occur frequently and exceptions are numerous. A just-in-case approach where the designer specifies all possible paths in the process model is impractical, leading to models that are large and unintelligible. Applications operating in such environments may be better served by a just-in-time approach, where adaptation and personalization may be done after the process has been deployed and without requiring all executions to perfectly align with the process model.

Existing methods and techniques in the area of adaptive, dynamic, and flexible workflow systems have addressed issues such as specifying exception handling mechanisms within process models [6, 13] or migrating running processes when replacing a previously deployed process model with a new one [1, 14]. However, these prior proposals do not provide mechanisms to alter the behaviour of process-oriented composite applications after deployment *without changing the process model*, that is, without requiring alignment between each execution of the composite application and its process model (whether the originally deployed model or a modified version of it). Such ad hoc flexibility mechanisms are instrumental for a number of purposes including: (i) personalising applications to suit the requirements or preferences of specific users; (ii) adapting the behaviour of composite applications based on the users' context (e.g. location, device or network connection) without overloading the process model with such details; and (iii) hot-fixing the composite application to address unforeseen errors, as opposed to predicted exceptions, or to add new features (e.g. to plug-in new applications or to re-route tasks and data).

To overcome the above limitations of existing systems, we propose to adopt an event-based coordination approach to execute process-oriented composite applications. Due to its finer-grained nature, event-based coordination approaches has several advantages over process-based ones when it comes to runtime adaptation and re-configuration [12]. By translating process models of composite applications into event-based models and using the latter in the runtime environment, it becomes possible by adding and removing event-based rules (e.g. event subscriptions related to a specific task) to overlay behaviour on top of already deployed composite applications in response to special requirements or unforeseen situations. In this way, users, administrators and/or developers can re-route data and control in an already deployed composite application in order to steer it into executions paths not foreseen in the process model, thereby facilitating the personalization and adaptation of these applications.

The main contribution of this paper is a technique for translating a process model described in a mainstream process modelling notation (UML Activity Diagrams) into an event-based model described through coordination rules made up of composite event specifications, predicates, and a small number of publishing/sharing primitives. We also discuss how the resulting event-based model can be executed on top of a shared object space infrastructure and how adaptation and personalization is achieved by adding rules (encoded as active objects) into the shared space. We illustrate the proposed technique and its supporting infrastructure through a use case scenario drawn from the area of mobile computing, where the need for adaptation and personalization is often prominent.

The paper is structured as follows. First, we outline a use case scenario (Section 2) and describe the coordination primitives and infrastructure upon which our proposal relies (Section 3). In Section 4 we introduce a technique for translating a process model captured as a UML activity diagram into an event-based coordination model. We then discuss how adaptation can be achieved by adding, enabling and disabling rules in the event-based model (Section 5). Finally, we discuss related work (Section 6) and conclude (Section 7).

2 Use Case Scenario

This section presents a use case that will be used as a motivating and working example in the rest of the paper. The scenario is described as a UML activity diagram[3] in Figure 1. We chose UML activity diagrams as a process modelling notation because of its status as a *de jure* standard and because its constructs are representative of those found in other process modelling and process execution languages (e.g. sequence, fork, join, decision and merge nodes). Thus the proposed techniques can be adapted to other languages that rely on these constructs. Moreover, a recent study shows that UML activity diagrams (version 2.0) provide direct support for many common workflow patterns [17].

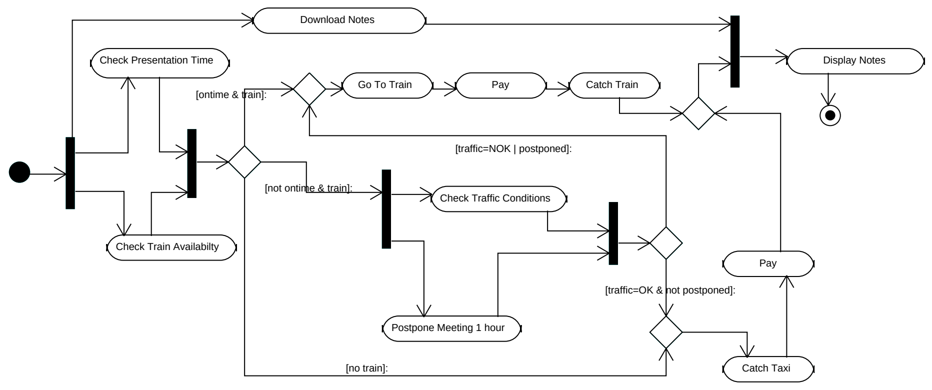

Fig. 1. UML Activity Diagram describing the working example

The scenario is an example of a *personal workflow* [11], i.e. a process aimed at assisting a user in the achievement of a goal that requires the execution of a number of tasks. Most of the tasks composing the process (but not necessarily the

[3] http://www.uml.org. Note that in this paper, we refer to UML version 2.0

process itself) are intended to be executed in a mobile device. Thus the scenario is also an example of a *mobile workflow*. We have chosen this scenario because, putting aside their futuristic nature, mobile and personal workflows constitute a class of process-oriented composite applications in which personalization and runtime adaptation are prominent requirements. Such requirements can also be found to varying degrees in more traditional applications (e.g. order handling) and the proposed techniques are also applicable in these settings.

In this scenario, a user is on a trip to attend a meeting. Before the meeting commences he runs this process-oriented application so that it assists him in the lead-up to the meeting. The process starts with three activities in parallel: 1) checking the presentation time, 2) checking the availability of trains to the destination, and 3) downloading meeting notes to the user's device (which may take some time due to low bandwidth).

After the presentation time and the train availability of the train have been checked three options are available: 1) If the user is "on time" AND "there is a train" that would take the user near the meeting's location, the user is directed to the train station; 2) If there is "no train", a taxi is automatically ordered; 3) If the user is "late" AND "there is a train", two new activities are started to determine if a taxi or a train is the best option for the user. At this point, the process checks the traffic conditions and tries to postpone the meeting by one hour (both actions in parallel). If the traffic is adverse, there is no point in catching a taxi, and the application will advise the user to catch the train. The same applies if the meeting is postponed. If however, there is favorable traffic and the meeting can not be postponed, the user will catch a taxi to get there sooner. Each transportation requires a payment. Payment is automatically arranged by the application and the details of the payment are sent to the finance department to arrange for a refund (both of these steps are modelled as a single task "pay"). Finally, once all the user is on his/her way to the meeting and the meeting notes have been downloaded, the application displays the notes.

3 Infrastructure for event-based model execution

This section presents the coordination infrastructure upon which the proposed technique relies and the framework to describe event-based coordination models.

3.1 The Active Object Space

To be able to execute the event-based coordination models that will be derived from process models, we require an execution infrastructure with support for : (i) event publishing, data transfer/sharing, and complex event subscription; (ii) association of reactions to event occurrences; and (iii) runtime re-configuration so that new event subscriptions and reaction rules can be added anytime. For reasons outlined below, we have chosen the *Active Object Space* (AOS) [7, 8] as our target infrastructure. The AOS is an exemplar of a family of communication infrastructure known as *coordination middleware* which has its roots in the tuple space model underlying the Linda system [10]. Other exemplars of coordination middleware include Sun's JavaSpaces[4] and IBM's TSpaces[5].

At the centre of the AOS is a shared memory (the *space*). Coordination between applications occurs through objects being written and taken from the space. Some of these objects may correspond to data that needs to flow from one application to another, while others may serve as signposting, indicating that a given step of work has been completed or that a given step of work is enabled but has not yet started. The AOS supports undirected decoupled communication based on four elementary operations, namely *read*, *write*, *take* and *notify*. A read operation copies an object from the space matching a given object template; a take operation moves an object matching a given object template out of the space; a write operation puts an object on the space; and a notify operation registers a subscription for a composite event expressed as a set of object templates: Whenever there is a combination of objects present in the space that matches these object templates, an event occurrence will be raised and a notification will be sent to the subscriber. An *object template* is an expression composed of a class name and a set of equality constraints on the properties of that class. An object matches a template if its class is equal to or is a sub-class of the class designated by the template and it fulfills the template's constraints.

An originality of the AOS with respect to other object-oriented coordination middleware lies in its support for *active objects*, that is, objects with their own thread of control that run on the space. Active objects can be deployed, suspended, resumed, and destroyed by applications running outside the space at any time. Active objects can read and write passive objects to/from the space, subscribe to events, and receive notifications from the space. At the implementation level, the difference between active objects and "passive" objects is that an active object has a special *execute* method that is invoked on a dedicated thread of control when the object is written into the space.

As illustrated in the rest of the paper, the deployment of active objects operating on a shared memory and writing and taking objects to/from this space, constitutes a powerful paradigm not only for executing event-based coordination models, but also for re-configuring these models after their deployment. Reconfiguration is facilitated by two features of the AOS infrastructure: (i) the use of undirected (also known as "generative") communication primitives which allows data and events to be produced and consumed without a priori determined recipients (and thus allows data and control to be "re-routed"); and (ii) the ability to add, remove, suspend and resume individual active objects and thus alter the behaviour of an application.

This having been said, we recognize that other coordination middleware or publish/subscribe middleware supporting composite events (e.g. Elvin[6]) constitute suitable alternatives to the AOS. To adapt our proposal to such alternative infrastructures, active objects would have to be replaced by dedicated applica-

[4] http://java.sun.com/developer/products/jini/index.jsp
[5] http://www.almaden.ibm.com/cs/TSpaces
[6] http://elvin.dstc.edu.au

tions operating outside the space (or operating on top of the messaging bus in the case of a pub/sub middleware).

3.2 Coordinators

Having introduced the basic concepts and functionality of the AOS, we now define a higher-level concepts that we use to explicate the execution of event-driven coordination models.

A *coordinator* is an active object that is deployed in the space to coordinate work (e.g. to perform synchronization or data transfer) and operates in an infinite loop until suspended or destroyed, with each iteration comprising three phases:

1. Waiting for an event, which could be either the addition to the space of an object matching a given template or an interaction initiated by an external application;
2. Performing internal processing and/or interacting with external applications;
3. Writing one or several objects to the space.

For methodological reasons, it is useful to distinguish two types of coordinators, namely *connectors* and *routers*. This way, internal coordination steps within the space (which is the responsibility of the routers) are separated from communication with external applications (which is the responsibility of the connectors). The following paragraphs explain these types of coordinators in turn.

Connectors A *connector* is a type of coordinator dedicated to enabling a connection between the space and one or several external applications. Connectors are necessary because external applications will generally not be programmed to interact with the space but will instead they rely on other communication protocols and interfaces. Thus, a way of wrapping external applications so that they can be coordinated through the space is necessary and this is what connectors achieve. For example, a connector could be placed on the space for the purpose of relaying context data between a sensor and the space. This connector would: (i) receive or poll data from the sensor; (ii) encode these data as a passive object; and (iii) write this object in the space, possibly overriding the object containing the previous known state of the context data. Another example of a connector is an active object that calls an external web service when an object of a certain type is written to the space, like for example an object that indicates that a certain task has completed. This latter example shows that connectors can be used as a mechanism to detect that a given task is enabled and thus that a given application has to be invoked to perform this task.

Control routers Control routers (or routers for short) react to the arrival of an object or a combination of objects to the space and perform some processing before producing a set of new objects and writing them onto the space. The processing that a router performs is generally translation of data using a specified operation. This can be a simple operation such as an arithmetic operation, or more complex operations such as checking that a purchase order is valid, but in any case, this operation should not involve interaction with external applications, since interactions with external applications are handled by the connectors.

A router is described by the following elements:

- Input set: A set made up of a combination of object templates and boolean conditions.
- Output: A set of expressions, each of whichs evaluates into an object.
- Stop set: A set containing a combination of object templates and boolean conditions.
- Replace set: A set of coordinators.

The way these elements are used is as follows. Upon creation, the router will place a subscription with the space for the set of object templates contained in its input set (i.e. the set obtained after removing the boolean conditions from the input set). Subsequently, the router will be notified whenever a set of objects matching these templates are available on the space. At this point, the router evaluates the set of conditions in its input set. If all these conditions are true, the router proceeds to “take” the set of objects in question and if it succeeds to take them, it will evaluate the transformation functions (i.e. the expressions in the “Output”) taking these objects as input. The objects resulting from the transformation are then written back to the space. The “input set” thus captures the events and conditions that lead to the activation of a router (where an event corresponds to the arrival of an object to the space). The “Output” on the other hand encodes the events that the router will produce upon activation, i.e. the objects to be placed in the space for consumption by other coordinators. Finally, if a set of objects matching the object templates in the stop set is found on the space, the router will terminate its execution and replace itself by the set of routers specified in the replace set.

A set of routers can be deployed and interconnected with existing applications (through connectors) in order to coordinate the execution of the instances of a process. During the execution of a process instance, routers read and take from the space, objects denoting the completion of tasks (i.e. *task completion objects*) and write into the space objects denoting the enabling of tasks (i.e. *task enabling objects*). Connectors on the other hand read and take task enabling objects, execute the corresponding task by interacting with external applications, and eventually write back task completion objects, which are then read by routers. To make sure that routers only correlate task completion events relating to the same instance of a process, every object template in the input set of the router will contain a constraint stating that all the matched task completion objects must have the same value for the attribute corresponding to the process instance identifier (*piid*). In addition, when a router (connector) writes a task enabling (task completion) object to the space, it includes the corresponding piid. A process instance is created when a “process instantiation” object with the corresponding process and process instance identifier is placed on the space by a connector. It is the responsibility of the connectors which place such objects to ensure that process instance identifiers are unique.

4 From process-based to event-based models

This section focuses on the issue of generating coordinators for process orchestration from UML activity diagrams. We first describe the technique for generating coordinators from UML activity diagram restricted to control-flow constructs. We then show how data-flow aspects are incorporated.

4.1 Translating control-flow constructs into input sets

For each action[7] in an activity diagram, a connector will be generated to handle its execution, which in the case of process-oriented composite applications will involve an interaction with an external application. Connectors thus encapsulate the execution of actions in the process.

On the other hand, a number of routers are generated for each action. The input sets for these routers are generated according to the algorithm sketched using a functional programming notation in Figure 2 and explained below. The main function defined by this algorithm (namely AllInputSets) takes as input an activity diagram represented as a set of nodes (action, decision, merge, fork, join, initial, and final nodes) inter-linked through transitions. From there, it generates a set of input sets (see definition of input set in Section 3.2). The input sets produced by this algorithm can then be used to create a collection of routers (one router per input set) that collectively are able to coordinate the execution of instances of the process in question. Intuitively, each input set encodes one possible way of arriving to a given node in the process.

Given the set of connectors and routers deployed for a process-oriented composite application, execution occurs as follows. A router corresponding to an action node will wait until the object templates in its input set are all matched, at which point if all the boolean expressions in the input set evaluate to true, it will place an object on the space to indicate that the action is enabled and thus that the corresponding external application invocation may be performed by a connector. Once the connector has completed its interaction with the external application, it will put an object in the space to signify this completion. Such *completion objects* will then match the object templates of the input set of another router, eventually causing the activation of this other router. In this way the execution of the process moves from a router corresponding to a given action, to another. The initial and final states are mapped trivially to two routers that respectively detect the commencement of the process instance and perform clean-up (i.e. delete all remaining objects related to the completed instance).

Algorithm for input sets generation The algorithm focuses on a core subset of activity diagrams covering only initial and final nodes, action nodes, and control nodes (i.e. decision, merge, fork, and join nodes) connected by transitions. In particular, the algorithm does not take into account object flow (which is discussed later) nor swimlanes (which are irrelevant for the purposes of this paper).

[7] Action is the term used in UML activity diagrams to refer to a “task”.

Without loss of generality, the algorithm assumes that all conditional guards in the activity diagram are specified in disjunctive normal form. Also without loss of generality, the algorithm assumes that there are no "implicit" forks and joins in the diagram. An implicit fork (join) occurs when several transitions leave from (arrive to) an action node. In this case, the semantics of this fragment of the diagram is the same as that of a diagram in which this action node only has one outgoing (incoming) transition leading to (originating from) a fork node (a join node). Thus implicit forks and joins should be eliminated from a diagram and replaced by explicit fork and join nodes prior to applying this algorithm.

```
AllInputSets(p: Process) :
        let $\{x_1, \ldots, x_n\}$ = ActionNodes(p) in
           InputSets($x_1$) $\cup \ldots \cup$ InputSets($x_n$)
InputSets(x : Node) :
     let $\{t_1, \ldots t_n\}$ = IncomingTrans(x) in
        return InputSetTrans($t_1$) $\cup \ldots$ InputSetTrans($t_n$)

InputSetsTrans(t : Transition) :
     let x = Source(t)
        if NodeType(x) = "action"
          return CompletionObject(x)
        else if NodeType(x) = "initial"
               return ProcessInstantiationObject(Process(x))
        else if NodeType(x) $\in$ {"decision", "fork"}
               let $\{c_1, \ldots, c_n\}$ = Disjuncts(Guard(t)),
                   $\{i_1, \ldots, i_n\}$ = InputSets(Source(t)) in
                   return $\{\{c_1\} \cup i_1, \ldots, \{c_1\} \cup i_n\}$,
                          ...
                          $\{c_n\} \cup i_1, \ldots, \{c_n\} \cup i_n\}$
        else if NodeType(x) = "merge"
               let $\{t_1, \ldots, t_n\}$ = IncomingTrans(x) in
                   return InputSetsTrans($t_1$) $\cup \ldots \cup$ InputSetsTrans($t_n$)
        else if NodeType(x) = "join"
               let $\{t_1, \ldots, t_n\}$ = IncomingTrans(x),
                   $\{\langle i_{1,1}, \ldots, i_{1,n}\rangle$,
                    ...
                    $\langle i_{m,1}, \ldots, i_{m,n}\rangle\}$ =
                   InputSetsTrans($t_1$) $\times \ldots \times$ InputSetsTrans($t_n$) in
                   return $\{i_{1,1} \cup \ldots \cup i_{1,n}$,
                          ...
                          $i_{m,1} \cup \ldots \cup i_{m,n}\}$
```

Fig. 2. Algorithm for deriving input sets from an activity diagram.

Figure 2 defines three functions: the first one, namely AllInputSets generates all the input sets for a process by relying on a second function, namely InputSets, which generates a set of input sets for a given node of the diagram. This latter function relies on a third (auxiliary) function named InputSetsTrans, which produces the same type of output as InputSets but takes as parameter a transition rather than a set. This definition of InputSetsTrans operates based on the node type of the source of the transition, which may be an action node, an initial node, or one of the four types of control nodes. If the transition's source is an action node, a single input set is returned containing a completion object (see Section 3.2) for that action. Intuitively, this means that the transition in question may be taken when a completion object corresponding to that action is placed on the space. Similarly, if the source of the transition is the initial node of the activity diagram, a single input set with a "process instantiation" object is created, indicating that the transition in question will be taken when an object is placed on the space signalling that a new instance of the process must be started. If the transition's source is a control node, the algorithm keeps working backwards through the diagram, traversing other control nodes, until reaching action nodes. In the case of a transition originating from a decision or a fork node, which is generally labelled by a guard (or an implicit "true" guard if no guard is explicitly given), the transition's guard is decomposed into its disjuncts, and an input set is created for each of these guards. This is done because the elements of an input set are linked by an "and" (not an "or") and thus an input set can only capture a conjunction of elementary conditions and completion/instantiation objects (i.e. a disjunct). Finally, in the case of a transition originating from a "merge" (resp. a "join"), the function is recursively called for each of the transitions leading to this merge node (join node), and the resulting sets of input sets are combined to capture the fact that when any (all) of these transitions is (are) taken, the corresponding merge node (join node) may fire.

The following notations are used in the algorithm:

- ActionNodes(p) is the set of action nodes contained in process p (described as an activity diagram).
- Source(t) is the source state of transition t
- Guard(t) is the guard on transition t
- Disjuncts(c) is the set of disjuncts composing condition c
- IncomingTrans(x) is the set of transitions whose target is node x
- NodeType(x) is the type of node x (e.g. "action", "decision", "merge", etc.)
- Process(x) is the process to which node x belongs.

Example Figure 3 describes the router for the "CheckTraffic" using a concrete XML syntax. This action node will only have one router associated to it because there is only one path leading to the execution of this action. Indeed, to execute this action, it is necessary that both the "check presentation time" and the "check train availability" actions have completed, and in addition that the condition "not ontime and train" evaluates to true, and this condition does not contain any disjunction. When all these conditions are satisfied, the router will produce an enabling object that will eventually be picked up by the connector associated to action "check traffic".

It can be noted in this example that the process instance identifier (piid) attribute of the completion object templates are associated with a variable. In

```
<Router name = ‘‘CheckTrafficEnabler’’>
  <Input>
   <Template>
   <CompletionObject actionName=’’CheckPresentationTime’’ piid=’’var:X’’/>
   </Template>
   <Template>
  <CompletionObject actionName=’’CheckTrainAvailability’’ piid=’’var:X’’/>
   </Template>
   <Condition>
    <Equality variable=’’ontime’’ value=’’false’’/>
   </Condition>
   <Condition>
    <Equality variable=’’train’’ value=’’true’’/>
   </Condition>
  </Input>
  <Output>
   <EnablingObject action=’’CheckTraffic’’ piid=’’var:X’’/>
  </Output>
</Router>
```

Fig. 3. Sample router

the concrete XML syntax, an XML namespace (aliased "var") is reserved to refer to variables. The AOS is capable of interpreting collections of object templates where some of the atttributes are associated with such variables and to match these templates in a way that if the same variable is associated with attributes of two different templates, then the objects matching these templates should contain the same values for these attributes.

4.2 Incorporating Data-Flow

Data flow (or more precisely *object flow*) in activity diagrams is represented by object nodes, represented as rectangles as illustrated in Figure 4. Object nodes are directly linked to a "producing" action preceding the object node. They are also linked, either directly or through the intermediary of a number of control nodes, to one or several "consuming" action node(s) following the object node. In the example of Figure 4, the user pays using his mobile device and this produces a receipt object which is then forwarded to the finance department so that the user may obtain a refund.

In terms of the proposed technique, object flows are treated as follows. The production of objects for a given object node is the responsibility of the connector corresponding to the action node directly preceding this object node (i.e. the producing action). In other words, the corresponding object would appear as one of the elements in the "output" of this coordinator (see Section 3.2). In the example at hand, the production of objects of type "Receipt" is done by the connectors of the action nodes labelled "Pay". On the other hand, the consumption of objects corresponding to an object node is carried out by the

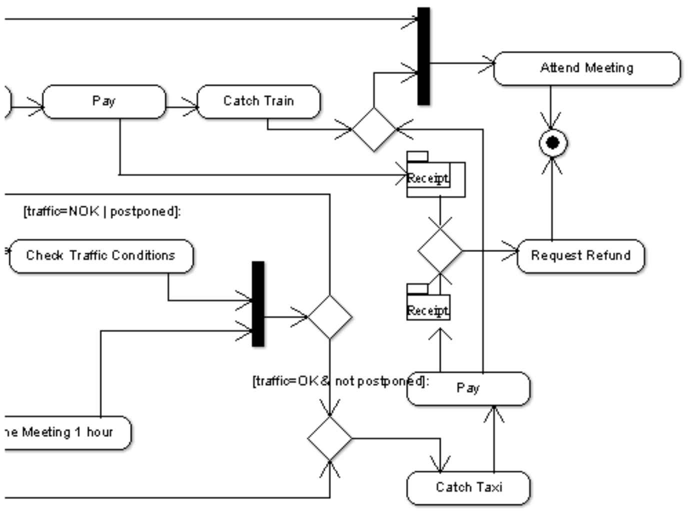

Fig. 4. Working example with object nodes

connectors of action nodes that follow this object node, either directly or through the intermediary of a number of control nodes (i.e. the consuming actions). In the example at hand, this means that the connector of the action node labelled “Request Refund” will take an object of type “Receipt” from the space when this action is enabled.

Since object flow is handled exclusively by connectors, the algorithm presented above does not have to deal with object nodes. Accordingly, object nodes should be removed from the activity diagram before applying the algorithm for deriving input sets. Removing object nodes from an activity diagram is trivial since they always have only one incoming and one outgoing transition.

5 Achieving adaptation

In certain situations, some functionality may or should be made unavailable. A context change may mean that some processing can not be performed, or a user moving outside a firewall may prevent him/her from executing certain applications. In our working example, it may happen that the system takes too much time to contact the other meeting participants to check if the meeting can be postponed (i.e. the execution of the “postpone meeting” may take more time than the user is willing to wait for). In this case, a user may indicate that (s)he does not wish to be delayed by this action, but instead, if the “Check Traffic” action is completed and if the traffic conditions are OK, (s)he would immediately take a taxi. This adaptation can be achieved by activating the router specified in a concrete XML syntax in Figure 5. In this XML fragment, we assume that the piid of the process instance for which this modification is to be done is 1. The element StopSet indicates that this router is disabled if the “Postpone Meeting” action is completed. Thus this router will only place an enabling object to trigger action “Catch Taxi” if the action “Check Traffic” completes before “Postpone Meeting” and the corresponding boolean expression evaluates to true.

```
<Coordinator name = ‘with participants’’>
  <Input>
   <Template>
    <CompletionObject action=’’CheckTraffic’’ piid=’’1’’/>
   </Template>
   <Condition>
    <Equality variable=’’traffic’’ value=’’OK’’/>
   </Condition>
  </Input>
  <Output>
     <EnablingObject action=’’CatchTaxi’’ piid=’’1’’/>
  </Output>
  <StopSet>
    <CompletionObject action=’’PostponeMeeting’’ piid=’’1’’/>
  </StopSet>
</Coordinator>
```

Fig. 5. Sample router for process adaptation

The above adaptation could arguably be achieved by modifying the process model.[8] However, in this case, significant tool support would be required and model versioning may become an issue. In contrast, enabling an event-based rule (encoded as a router) provides a more lightweight adaptation mechanism.

More radical changes may also be made. For example, consider a user that prefers taxis over trains in any case and so would always catch taxis regardless of traffic conditions and amount of time before the meeting. In this case, a router may be introduced that enables the action “CatchTaxi” immediately upon process instantiation when the process instance is started by the user in question. At the same time, all other routers for that process instance would be disabled, except the ones for download notes and display notes.

How the user actually specifies “dynamic” changes to composite applications is a user interface issue outside the scope of this article. This may be achieved, for example, by means of personalization applications running as active objects and disabling or enabling routers or placing completion or enabling objects according to an adaptation logic previously coded by a developer. Another option is to provide users with options for adapting/personalising applications. When a user manually selects one of these options, a number of coordinators are enabled and/or completion and enabling objects are written to or taken off the space.

[8] Note that expressing this type of discriminator (or 1-out-of-2) join in UML activity diagrams requires the use of advanced constructs (namely signals) not covered by our algorithm [17]. However, this is not the point that we try to make here.

Of course, this mechanism may be abused and lead to undesirable effects such as deadlocked executions. However, as shown above, adaptation may be scoped to specific process instances to avoid affecting a wider user base. In addition, as certain adaptations become permanent, they may be propagated back to the process model resulting in a new process model being deployed.

6 Related Work

Process-oriented application development has been the subject of significant attention in the last decade, prompting the emergence of a large number of process modelling and execution languages, some of which have been the subject of standardisation initiatives such as the Business Process Execution Language for Web Services (BPEL4WS).[9] However, the platforms supporting these languages adopt an approach to process-oriented application development that is not suitable in scenarios where personalization and adaptation are prominent requirements. Indeed, these platforms typically rely on the static definition of process models and allow little change to occur without a significant redeployment effort.

As discussed in the Introduction, proposals in the area of adaptive and flexible workflow [14] generally focus either on a priori adaptation (e.g. attaching exception handling policies to a process model) or on dealing with changes in the process model. In contrast, we advocate that adaptation should not be handled at the level of the process model. Our proposal shows that if an event-based coordination model is used at the execution layer, it is possible to make fine-grained changes to specific parts of the process and to confine these changes to specific process instances, without altering the process model. In other words, the process model can be used as a reference to deal with the majority of cases but deviations can occur for specific cases based on the activation or de-activation of the rules composing the event model. Parallels can be drawn between our approach and the one followed in case handling systems [3] where human workers route cases (i.e. process instances) manually based on information associated to each case and contextual information such as workload and resource availability. However, case handling is targeted at processes composed mostly of manual tasks. In contrast, our proposal is targeted at processes in which tasks are delegated to software applications so that it is not possible to count on human intervention at each step of the process.

There exist a large body of proposals in the area of coordination architectures, and in particular space-based ones. Some of these architectures (e.g. Mars [5] and Limone [9]) support the definition of reaction rules to coordinate application components, similar to the way coordinators operate in our framework. However, despite their potential synergies, proposals in the areas of coordination architectures on the one hand, and process-oriented application development on the other, have so far evolved independently – a notable exception being the work by Tolksdorf [16] who describes a space-based architecture for routing XML

[9] `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`

documents through processing steps encoded in XSL. A major novelty of our proposal is that it seamlessly combines techniques from coordination-based and from process-oriented software architectures.

This paper partly builds upon previous work on decentralised orchestration of process-oriented composite services specified as UML statecharts [4]. In this prior work, an algorithm was proposed that bears some similarities with the one presented in Figure 2. In addition to technical differences between the algorithms, stemming in part from the use of activity diagrams (version 2.0) rather than statecharts, the proposal of this paper differs from the previous one in the use that it makes of the output of the algorithm: Instead of using this output for decentralised orchestration, it uses it for event-based centralised orchestration based on coordination middleware. The proposal in this paper can also be seen as a refinement of the architecture presented in [15], where agents and tuple spaces are combined in an architecture for service composition. In the present paper, we have presented a concrete approach to encode and execute event-based models and we have detailed a method for generating event-based models from process-based ones. We have also shown that by encoding event-based models as active objects it is possible to achieve various forms of adaptation.

7 Conclusion and Future Work

This paper has shown how a process model specified using UML activity diagrams can be translated into an event-based model that can be executed on top of a coordination middleware. Specifically, a process model is encoded as a collection of active objects that interact with each other through a shared object space. We have argue and illustrated that this approach is suitable for undertaking post-deployment adaptation of process-oriented composite applications. In particular, new control dependencies can be encoded by dropping new (or enabling existing) active objects into the space and/or disabling existing ones.

A possible direction for future work is to extend the proposed algorithm for input sets generation to cover a larger set of process modelling constructs such as signals in UML activity diagrams or advanced control-flow constructs such as those found in YAWL [2]. Another direction for future work is to design a mapping from event-based models to process models. The idea would be to automatically derive a process model from a collection of routers. This "reverse" mapping would assist developers in propagating changes in the event-based model to the process model, when it is decided that these changes should be made permanent. Techniques such as those developed in the setting of process mining, where process models are derived from causal relations extracted from execution traces, could provide insights for designing this reverse mapping.

Acknowledgments The first author is funded by a Queensland Government Smart State Fellowship co-sponsored by SAP. The second author is funded by an SAP-sponsored scholarship.

References

1. W. M.P. van der Aalst. How to handle dynamic change and capture management information: An approach based on generic workflow models. *Computer Systems Science and Engineering*, 15(5):295–318, 2001.
2. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2004.
3. W.M.P. van der Aalst, M. Weske, and D. Grünbauer. Case handling: A new paradigm for business process support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.
4. B. Benatallah, M. Dumas, and Q.Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases*, 15(1):5–37, January 2005.
5. G. Cabri, L. Leonardi, and F. Zambonelli. Reactive tuple spaces for mobile agent coordination. In *Proceedings of the Second International Workshop on Mobile Agents (1998)*, pages 237–248, Stuttgart, Germany, 1999. Springer Verlag.
6. F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems*, 24(3):405–451, 1999.
7. K. Elms, S. Milliner, and J. Vayssiere. Object spaces with active objects. U.S. Patent Application # 2004P00851US, filed 29 December 2004.
8. T. Fjellheim, S. Milliner, M. Dumas, and K. Elms. The 3DMA middleware for mobile applications. In *Proceedings of the 2004 International Conference on Embedded and Ubiquitous Computing*, Aizu, Japan, August 2004. Springer Verlag.
9. C-L. Fok, G-C. Roman, and G. Hackmann. A lightweight coordination middleware for mobile computing. In *Proceedings of the 6th International Conference on Coordination Models and Languages*, pages 135–151, Pisa, Italy, February 2004. Springer Verlag.
10. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming*, 2(1):80–112, January 1985.
11. S-Y. Hwang and Y-F. Chen. Personal workflows: Modeling and management. In *Proceedings of the 4th International Conference on Mobile Data Management*, 2003.
12. D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, 2002.
13. R. Muller, U. Greiner, and E. Rahm. AgentWork: a workflow system supporting rule-based workflow adaptation. *Data and Knowledge Engineering*, 51(2):223–256, November 2004.
14. S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.
15. Q.Z. Sheng, B. Benatallah, Z. Maamar, M. Dumas, and A.H.H. Ngu. Enabling personalized composition and adaptive provisioning of web services. In *Proceedings of the International Conference on Advanced Intormation Systems Engineering*, pages 322–337, Riga, Latvia, June 2004. Springer Verlag.
16. R. Tolksdorf. Coordination technology for workflows on the web: Workspaces. In *Proceedings of the 4th International Conference on Coordination Models and Languages*, pages 36–50, Limassol, Cyprus, September 2000. Springer Verlag.
17. P. Wohed, W. M.P. van der Aalst, M. Dumas, A. H.M. ter Hofstede, and N. Russell. Pattern-based Analysis of the Control-flow Perspective of UML Activity Diagrams. In *Proceedings of the International Conference on Conceptual Modelling (ER)*, Klagenfurt, Austria, October 2004. Springer Verlag.