Pattern-Based Analysis of the Control-Flow Perspective of UML Activity Diagrams[*]

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

[1] Centre de Recherche en Automatique de Nancy, Université Henri Poincaré - Nancy 1/CNRS,
BP239, 54506 Vandoeuvre les Nancy, France
`petia.wohed@cran.uhp-nancy.fr`
[2] Department of Technology Management, Eindhoven University of Technology,
GPO Box 513, NL5600 MB Eindhoven, The Netherlands
`w.m.p.v.d.aalst@tm.tue.nl`
[3] Faculty of Information Technology, Queensland University of Technology,
GPO Box 2434, Brisbane QLD 4001, Australia
`{m.dumas, a.terhofstede, n.russell}@qut.edu.au`

Abstract. The Unified Modelling Language (UML) is a well-known family of notations for software modelling. Recently, a new version of UML has been released. In this paper we examine the Activity Diagrams notation of this latest version of UML in terms of a collection of patterns developed for assessing control-flow capabilities of languages used in the area of process-aware information systems. The purpose of this analysis is to assess relative strengths and weaknesses of control-flow specification in Activity Diagrams and to identify ways of addressing potential deficiencies. In addition, the pattern-based analysis will yield typical solutions to practical process modelling problems and expose some of the ambiguities in the current UML 2.0 specification [9].

Keywords: UML, Activity Diagrams, Workflow Patterns, YAWL.

1   Introduction

The Unified Modelling Language (UML), frequently referred to as a de facto standard for software modelling, has recently undergone a significant upgrade to a new major version, namely UML 2.0[1]. Being a multi-purpose language, UML offers a spectrum of notations for capturing different aspects of software structure and behaviour. One of these notations, namely Activity Diagram (AD), is intended for modelling computational and business/organisational processes.

If the UML AD notation is to be adopted as a standard for business process modelling, it should compare favourably with other notations in this space. In order to facilitate such a comparison a comprehensive analysis on UML AD has been performed

---

[1] `http://www.uml.org`

and the results of it are reported here. The goal of this analysis has been to evaluate the capabilities and limitations of UML AD.

Evaluating and comparing modelling notations, particularly in the area of business processes, is a delicate endeavour. Empirical evaluations in terms of case studies may lead to valuable insights, but the conclusions are difficult to generalise due to their restricted scope. Theoretical evaluations on the other hand rely heavily on the evaluation framework they utilize. For instance, evaluations in terms of ontologies, such as the Bunge Wand and Weber (BWW) ontology [6,10], lead to coarse-grained results since these ontologies are composed of highly general concepts whose pertinence and manifestation in the context of business processes have not yet been studied.

In order to provide a more fine-grained analysis we have chosen a specialised evaluation framework. It is constituted by the set of workflow patterns defined on `www.workflowpatterns.com`. While originally developed as an instrument to evaluate languages supported by workflow systems, these patterns have also successfully been used to evaluate languages for process-aware information systems development [5,16,15,8,18]. Initially restricted to the control-flow perspective (i.e. the ordering of activities in a process) [3] these patterns have recently been extended in accordance with Jablonski and Bussler's classification [7] to accommodate the data perspective (which deals with data transfer between activities) [12] and the resource perspective (dealing with the resource allocation for the execution of the activities within a process) [11]. Moreover, based on the workflow patterns framework, a workflow definition language called YAWL (Yet Another Workflow Language) has been designed [2] and implemented [1]. YAWL provides a reference formalisation for the control-flow patterns.

Hence, we motivate our choice for using the workflow patterns framework by arguing that it is 1) well tested, 2) provides a sufficient level of granularity for a deep analysis and 3) it is the most complete and powerful framework existing for evaluating the capabilities of a process modelling language. Moreover, using a framework which has been applied numerous times, will facilitate comparison between the analysed languages.

Accordingly, this paper reports the results of an evaluation of UML 2.0 AD in terms of the workflow patterns. Due to space limitations it presents the results from the evaluation of the control-flow perspective only[2]. The contributions of the paper are:

- The identification of some limitations in UML AD and recommendations for addressing these with minimal disruption to the current design of the language.
- Discussions on how to capture the patterns in UML AD which provide elements of reusable knowledge for process designers that encounter these patterns.
- An analysis of UML AD, e.g., pointing out ambiguities in the behaviour part of the current version of the specification [9].

An evaluation of UML AD version 1.4 in terms of these patterns has been previously reported [5]. However, while in UML 1.4, activity diagrams are based on statecharts, in UML 2.0 they have a semantics defined in terms of token flow inspired by (though not fully based on) Petri nets. Thus, the evaluation of UML 2.0 leads to different results

---

[2] For an evaluation of the data perspective see [17].

than the one for UML 1.4. Furthermore, an attempt at evaluating UML 2.0 AD using the workflow patterns has been conducted by White [15]. However, some of the results reported by White may be questioned as explained in the remainder of this paper.

The paper is organised as follows. Section 2 briefly introduces the UML 2.0 AD notation. Section 3 reports on the evaluation of UML AD in terms of the control-flow patterns. Finally, Section 4 summarises the results and concludes the paper.

2 Overview of UML 2.0 AD

In UML AD the fundamental unit of behaviour specification is the *Action*. "An action takes a set of inputs and converts them to a set of outputs, though either or both sets may be empty." [9], p. 229[3]. Actions may also modify the state of the system. The language provides a very detailed action taxonomy, where more than 40 different action types are specified. However, a deep discussion of them is outside the scope of this paper and in Figure 1a we only present the action types that we have found to be relevant to our evaluation. These are *Action*, *Accept Event*, *Send Signal*, and *Call Behavior Action*.
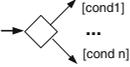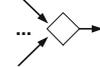
Action/Activity	AcceptEvent	InitialNode	ActivityFinal	FlowFinal	
CallBehaviorAction	SendSignal	Decision	Merge	Fork	Join
a) Actions			**b) Control Nodes**		

Fig. 1. UML 2.0 AD, Main Symbols

Furthermore, to present the overall behaviour of a system, the concept of *Activity* is used. Activities are composed of actions and/or other activities and they define dependencies between their elements. Graphically, they are composed of nodes and edges. The edges, used for connecting the nodes, define the sequential order between these. Nodes represent either *Actions*, *Activities*, *Data Objects*, or *control nodes*. The various types of control nodes are shown in Figure 1b.

3 Workflow Control-Flow Patterns in UML 2.0 AD

In this section, an analysis of UML AD version 2.0 is provided in terms of the control-flow patterns as defined in [3]. In this analysis YAWL (Yet Another Workflow Language) [2] is used as a reference realisation of the patterns (where appropriate). As YAWL is a formally defined language, its solutions for the patterns leave no room for ambiguity. Due to space restrictions the patterns themselves will not be discussed in detail here; for this the reader is referred to [3].

---

[3] In the remainder of this paper page numbers without reference refer to [9].

3.1   Basic Control-Flow Patterns, Multiple Choice and Multiple Merge

The first seven control-flow patterns, namely Sequence, Parallel Split, Synchronisation, Exclusive Choice, Simple Merge, Multiple Choice, and Multiple Merge are directly supported in UML AD. In fact, the first five of these patterns are supported by basically all process modelling and description languages and they correspond to control-flow constructs defined by the Workflow Management Coalition [14].

Figure 2 shows the solutions to the first seven patterns in UML AD and, as a point of comparison, in YAWL. The following paragraphs briefly discuss these solutions. Descriptions of the patterns are not included as they are relatively straightforward and they can be found in [3].

There are two ways of representing **Sequence** in YAWL (see Figure 2a). Two tasks can be connected directly, or they can have a condition (which corresponds to the concept of "place" in Petri nets) in between. Where two tasks are connected directly the condition in between exists in a formal sense, it is just not shown graphically. In UML, this basic pattern is solved in a very similar manner (see Figure 2b), i.e. through a control-flow arrow (p. 382), though UML does not explicitly support the notion of state hence there is no equivalent concept to the YAWL condition.

Fig. 2. Basic Control-flow Constructs in UML AD and in YAWL

The **Parallel Split** is captured in YAWL by an AND-split (see Figure 2c). In UML it is captured by a ForkNode, represented as a bar with one incoming edge and two or more outgoing edges (p. 404) (see Figure 2d). Furthermore, as "an action may have sets of incoming and outgoing activity edges..." (p. 336) and "when completed, an action execution offers [..] control tokens on all its outgoing control edges" (p. 337) the parallel split can also be modelled implicitly, by drawing the outgoing edges directly from the action node and omitting the fork node.

Synchronisation in YAWL is captured through the AND-join (see Figure 2e). In UML AD, the construct used for synchronisation is the JoinNode, i.e. a control node depicted as a bar with multiple incoming edges and one outgoing edge (p. 411) as shown in Figure 2f. A JoinNode may be associated with a condition (also called joinSpec). A joinSpec typically refers to the names of the incoming edges of the joinNode to which it is associated, but it may also be an arbitrary boolean expression. By default (i.e. if no joinSpec is provided as in Figure 2f), the joinSpec is taken to be an "and" of all the incoming edges, that is, a token has to be available at each of the incoming edges before the join node can emit a token, thus guaranteeing synchronisation of all incoming edges. Similarly to the parallel split solution, UML offers an "implicit" join notation: If the node that directly follows a join node is an action node, then the join node can be omitted and instead all the edges to be "joined" can be directly connected to the action node in question. The meaning of this implicit join is stated to be: "an action execution is created when all its [..] control flows prerequisites have been satisfied" (p. 337).

The **Exclusive Choice** in YAWL is captured by the XOR-split (see Figure 2g). In the YAWL environment, predicates specified for outgoing arcs of an XOR-split may overlap. In case multiple predicates evaluate to true, the arc with the highest preference (which is specified at design time) is selected. If all predicates evaluate to false, the default arc is chosen. The treatment of the XOR-split in YAWL guarantees that no matter what predicates are specified, exactly one outgoing branch will be chosen. In UML, a DecisionNode, graphically depicted by a diamond with one incoming edge and multiple outgoing edges, is used to represent this pattern (p. 387). The decision condition can be defined through "guards" attached to the outgoing edges (see Figure 2h). If the guard of more than one of the outgoing edges evaluates to true and if multiple edges accept the token and have approval from their targets for traversal at the same time, then the semantics of the construct depicted in Figure 2h is not defined (p. 388) and hence the "guards" should be made exclusive. A predefined "else" branch can be used which is chosen when none of the guards of the other branches evaluates to true. However, the use of "else" is optional.

In YAWL, the **Simple Merge** pattern is expressed using the XOR-join (see Figure 2i). In UML this pattern is represented by a MergeNode, that is, a diamond with several incoming edges and one outgoing edge (see Figure 2j). These solutions, for both YAWL and UML AD, also constitute a solution to the **Multiple Merge** pattern where parallelism may occur in the branches preceding the join and each completion of such a branch leads to (another) execution of the branch following the join.

In the **Multiple Choice** pattern, in contrast with the exclusive choice, zero, one, or multiple outgoing branches may be chosen. The multiple choice in YAWL is captured through the OR-split (see Figure 2k). It should be noted that in the YAWL environment

at least one outgoing branch will be chosen, which makes its OR-split slightly less general than the pattern. In YAWL, the selection of at least one branch is guaranteed by the specification of a default branch which is chosen if none of the predicates evaluate to true (including the predicate associated with the default branch). In UML AD, the solution for this pattern is the same as the solution for the parallel split, except that in addition guards controlling which branches should be started have to be defined for the edges departing from the ForkNode (see Figure 2l).

3.2   Synchronising Merge

Description. A form of synchronisation where execution can proceed if and only if one of the incoming branches has completed and from the current state of the process it is not possible to reach a state where any of the other branches has completed.

Solution in YAWL. The main challenge of achieving this form of synchronisation is to be able to determine where more completions of incoming branches are to be expected. In the general case, this may require a computationally expensive state analysis.

In YAWL a special OR-join symbol, directly captures this pattern (see the task D in Figure 3a and in Figure 3b). The semantics of the OR-join are such that it is enabled if and only if an incoming branch has signaled completion and from the current state it is not possible to reach a state where another incoming branch signals completion. While this can handle workflows of a structured nature like that in Figure 3a, it can also handle non-structured workflows such as the one displayed in Figure 3b.

As a possible scenario for the example in Figure 3b, consider the situation where after completion of activity A both activities B and C are enabled. Now, if activity C completes while activity B is still running, activity D has to wait for B's completion. More precisely, it has to wait for the outcome of the decision whether activity E or activity F shall be enabled (as activity B is an AND-split only one of the activities E and F will be enabled). If activity F is enabled, activity D has to wait for F's completion. If, instead, activity E is enabled (and it is not possible that any of the other outgoing branches of the OR-join will be enabled) then activity D is enabled as well.

For a more complete treatment of OR-joins in YAWL see [19].

Solution in UML. No direct support is provided for this pattern. White [15] provides a tentative solution. However, there are two problems with this solution. Firstly, it assumes the existence of a corresponding OR-split (i.e., as the example in Figure 3a),
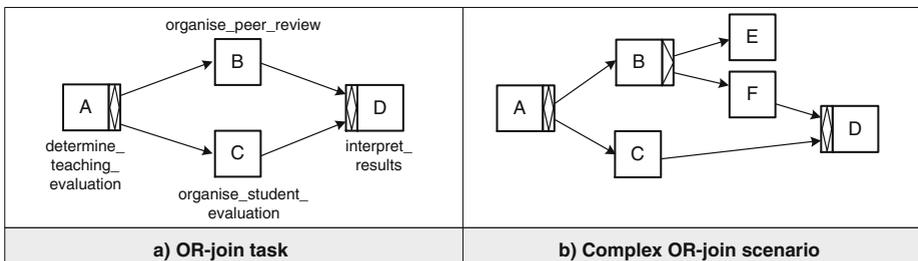

Fig. 3. Synchronising Merge in YAWL

hence it would not be general enough to work in an unstructured context (as exempli-fied in Figure 3b). Secondly, the solution proposed by White includes the following joinSpec: "a *condition expression* that controls how many Tokens must arrive from the incoming control flow before a Token will continue through the outgoing control flow" ([15], p. 11). This *condition expression* is, however, not specified and it is not clear how it could be determined how many tokens to expect. In addition, even if somehow this could be detected, how can one deal with multiple tokens arriving on the same branch as a result of loops?

3.3   Discriminator

Description. A form of synchronisation for an activity where out of a number of in-coming branches executing in parallel, the first branch to complete initiates the activity. When the other branches complete they do not cause another invocation of the activity. After all branches have completed the activity is ready to be triggered again (in order for it to be usable in the context of loops). The discriminator is a special case of the N-out-of-M Join (also called *partial join* [4]) as it corresponds to a 1-out-of-M Join.

Solution in YAWL. In YAWL, one of the ways to capture the discriminator involves the use of cancellation regions [2]. The discriminator is specified with a multiple merge and a cancellation region encompassing the incoming branches of the activity (see Figure 4a). In this realisation, the first branch to complete starts the activity in-volved, which then cancels the other executing incoming branches. This is not in exact conformance with the original definition of the pattern as it actually cancels the other branches. However, this choice is motivated by the fact that it is clear in this approach what the region is that is in the sphere of the discriminator giving it a clearer semantics.

Solution in UML. The solution in UML AD (see Figure 4c) uses the concept of Inter-ruptibleActivityRegion (p. 409) which is very similar to the notion of cancellation re-gion in YAWL. Hence, the solution is very close to the solution in YAWL. Furthermore, due to the use of weights (p. 352) on the interruptingEdge it also easily generalises to the N-out-of-M Join. YAWL provides direct support for N-out-of-M Join too, but the solution there is based on the concept of thresholds within the multiple instances task construct. This solution is shown in Figure 4b and the multiple instances task concept is further discussed in subsection 3.5.
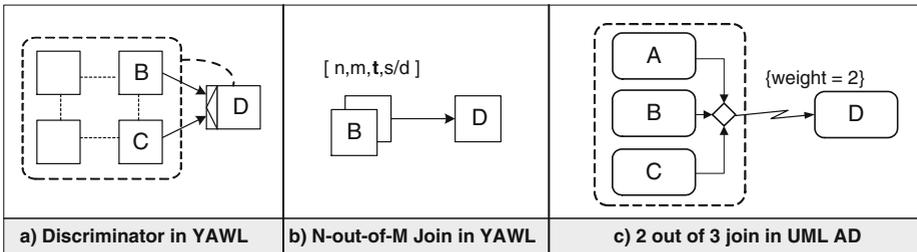

| a) Discriminator in YAWL | b) N-out-of-M Join in YAWL | c) 2 out of 3 join in UML AD |

Fig. 4. Solutions for the Discriminator pattern

White [15] presents a solution which uses an expression which checks for each incoming branch whether it has completed. He claims that the first token to arrive will progress the flow and the other tokens will not. The expression given seems to be an annotation which is not part of the UML AD notation. In addition, it is unclear how this would work if the discriminator is to be activated more than once (e.g. because it appears in a loop).

3.4   Structural Patterns

In this section we briefly consider the patterns involving arbitrary cycles and implicit termination.

Description of Arbitrary Cycles. Some process specification approaches only allow the specification of loops with unique entry and exit points. *Arbitrary cycles* are loops with multiple ways of exiting the loop or multiple ways of entering the loop.

Both YAWL and UML AD (also pointed out by White [15]) support arbitrary cycles.

Description of Implicit Termination. A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the subprocess and no other activity can be made active (and at the same time the subprocess is not in deadlock). This termination strategy is referred to as *implicit termination*.

Solution in YAWL. YAWL deliberately does not support implicit termination in order to force workflow designers to think carefully about workflow termination.

Solution in UML. UML AD provides direct support for this pattern. There are two notions for capturing termination namely, ActivityFinalNode and FlowFinalNode (see Figure 1). "A flow final destroys all tokens that arrive at it" (p. 403). It does not terminate the whole activity but only a flow within it. Implicit termination is then captured by ending every thread within an activity with a FlowFinalNode (same as in [15]).

3.5   Multiple Instances Patterns

This section focuses on the class of so-called "multiple instances" (MI) patterns. These patterns refer to situations where there can be more than one instance of a task active at the same time in the same case. The first of these patterns is concerned with the creation of multiple instances.

Description of MI without Synchronisation. Within the context of a single case (i.e., process instance) multiple instances of an activity can be created, i.e., there is a facility to spawn off new threads of control. Each of these threads of control is independent of other threads. Moreover, there is no need to synchronise these threads.

Solution of MI without Synchronisation in UML AD. Consider the UML AD example in Figure 5a which is taken from [9] (Figure 267, p. 404). This UML AD example provides a partial solution to the pattern. Instances of "Install Component" are "spawned-off" through a loop and the conditions associated with the DecisionNode will determine how many such instances will ultimately be created.

The next three patterns deal with the synchronisation of multiple instances. The first such pattern is **Multiple Instances with a Priori Design Time Knowledge** which can typically be supported by replicating the activity involved as many times as required.

This is possible in UML AD. The other two patterns deal with synchronisation of multiple instances where the number of instances is not known at design time.

The pattern **Multiple Instances with a Priori Runtime Knowledge** captures the situation when for one case an activity is enabled multiple times. The number of instances of a given activity for a given case varies and may depend on characteristics of the case or availability of resources, but is known at some stage during runtime, before the instances of that activity have to be created. Once all instances are completed some other activity needs to be started.

The pattern **Multiple Instances without a Priori Runtime Knowledge** is based on the previous pattern with the further complication that the number of instances to be created (and later on synchronised) are not know at any stage during runtime, before the instances have to be created. Even while some of the instances are being executed or have already completed, new ones can be created.

Solutions in YAWL. YAWL provides direct support for the multiple instance patterns. A multiple instance task in YAWL has four attributes: the minimum number of instances to be created; the maximum number; a threshold for continuation (where the semantics is that if all created instances have completed or the threshold has been reached the multiple instance task can complete); and an attribute with the possible values *static* and *dynamic* indicating whether or not it is possible to create new instances when a multiple instance task has been started (see Figure 4b).

Solution of Multiple Instances with a Priori Runtime Knowledge in UML AD. Here too we consider a UML AD solution taken from [9] (Figure 262, p. 401) as the basis for our discussion (see Figure 5b). In this example, the notion of ExpansionRegion, where the region consists of a single action, is used twice, once for $BookFlight$ and once for $BookHotel$. The small rectangles, divided into compartments and attached to a region, are meant to represent the input/output collections of elements, for the region. The action/s in the region is/are executed once for each element from the input collection ("or once per element position, if there are multiple collections" (p. 396)). "On each execution of the region, an output value from the region is inserted into an output collection at the same position as the input elements" (p. 395). The semantics of differing numbers in the inputs and their corresponding output collections is not clear. Furthermore, the way in which the multiple instances are executed, i.e., "parallel", "iterative", or "stream", is defined through an attribute of the ExpansionRegion node.
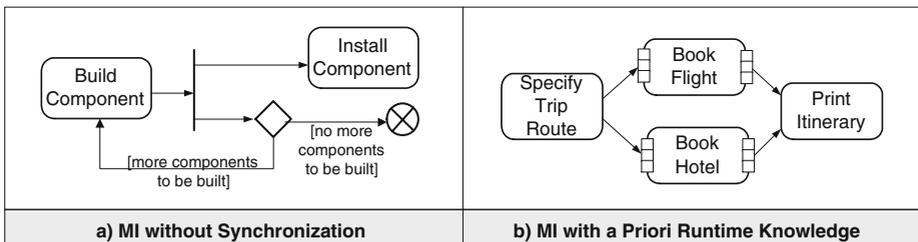

Fig. 5. Multiple Instances in UML AD (solutions reprinted from Figures 267 and 262 from [9])
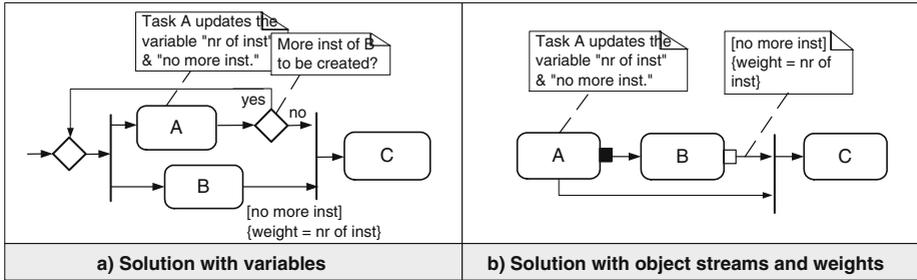
Fig. 6. MI without a Priori Runtime Knowledge in UML AD

The UML specification is not explicit about the completion of an ExpansionRegion, only about its initiation. We assume that it is completed when all its instances have completed.

Solution of Multiple Instances without a Priori Runtime Knowledge in UML AD. This pattern is not directly supported in UML AD. The notion of expansion region can not be used here as once an expansion region receives the required input collection(s) no values can be added afterwards. There are however workarounds that achieve the required functionality. The first solution, which is depicted in Figure 6a, is inspired by Figure 265, p. 403, from the UML specification [9] and by the solution provided by White [15]. The idea is to keep track of two variables, one representing the number of instances created so far, and one, a boolean, capturing whether there is a need to create more instances. The solution in Figure 6a is however more precise as to how synchronisation is to occur than both the solution provided by White [15] and the solution shown in [9]. Another workaround is to use object streams and weights. This solution is depicted in Figure 6b and exploits the fact that both the guard and the weight of an edge need to be satisfied (p. 352).

3.6 Deferred Choice

Description. A point in the process where one of several branches is chosen. In contrast to the XOR-split, the choice is not made explicitly (e.g. based on data or a decision) but several alternatives are offered to the environment (this is akin to the pick construct in BPEL4WS[4], the choice construct in BPML[5], or the event-based decision gateway construct in BPMN[6]). However, in contrast to the OR-split, only one of the alternatives is executed. This means that once the environment activates one of the branches, the other alternative branches are withdrawn. It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible.

Solution in YAWL. YAWL is based on Petri nets and therefore directly supports the deferred choice construct. A condition (the YAWL term for a place) is specified as

---

[4] `www-128.ibm.com/developerworks/library/specification/ws-bpel`
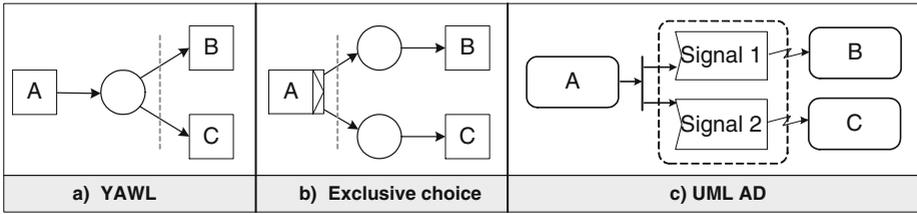[5] `www.bpmi.org`
[6] `www.bpmn.org`

Fig. 7. Deferred Choice (in UML AD the solution is identical to the one presented in [15])

input to the activities that can result from the choice. At runtime, the alternative that is chosen consumes the token thus disabling the other alternatives.

Figure 7a illustrates the solution and contrasts it with the solution of the Exclusive Choice pattern in Figure 7b. The vertical dotted lines drawn in these figures are only meant for emphasising the moment of choice and they are not part of the language.

Solution in UML. This pattern is captured in UML AD through a fork and a set of accept signal actions, one preceding each action in the choice. In addition, an interruptible activity region encircling these signals is defined (see Figure 7c). The semantics is that the first signal received will enable and trigger the activity following it (which follows from the definition of AcceptEventAction on p. 250) and disable the rest of the activities included in the deferred choice by terminating all other remaining receive signal actions in the region (which follows from the definition of InterruptibleActivityRegion on p. 409). This solution is identical to that proposed by White [15].

In UML AD 1.4 this pattern can be captured using a "waiting state" (see [5]), but this is not applicable in UML AD 2.0 due to its lack of support for the notion of state.

3.7 Interleaved Parallel Routing

Description. Several activities are executed in an arbitrary order: The order is decided at runtime, and no two activities are executed concurrently (i.e. no two activities are active for the same process instance at the same time).

Solution in YAWL. Given that YAWL is based on Petri nets, the idea of a mutex place can be used as presented in [3]. This solution is shown in Figure 8a. The finesse of this solution is that it is general enough to also capture the case where sequences of activities have to be interleaved. In the figure, the sequences to be interleaved are A, B with C, D. It is important that A is always executed before B and likewise, C before D (on top of the requirement that no two activities are executed at the same time).

Solution in UML. Similar to UML AD version 1.4, this pattern is not directly supported in UML 2.0 although a workaround solution can be designed using signals that act as semaphores. This is due to the absence of the notion of state (or the notion of "place" as supported in Petri nets). A workaround solution is shown in Figure 8b. Before an action can start this signal needs to have been received, and after the completion of an action this signal needs to be sent as to indicate that another action may now execute. In this solution, an action can start after the preceding AcceptEven action received the signal S. After it completes, the following action sends the signal again so that another
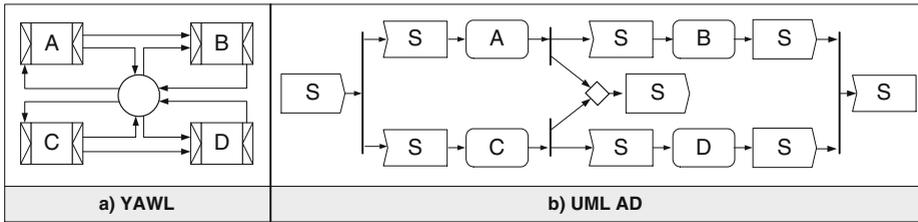
Fig. 8. Solutions for Interleaved Parallel Routing

action can be executed. As this other action may be in the same thread (e.g. after A it should be possible to execute not only C, but also B) there is a subtle issue of avoiding that an action of another thread will always "grab" the signal. This would occur if the SendSignal and the AcceptEvent actions were put in sequence, rather than in parallel, after completion of activities not last in a thread. The solution presented in Figure 8b assumes that 1) a signal can be sent from an action in a flow to an action in the same flow, 2) even though there may be multiple receivers ready to receive a signal only one of them will actually consume it (this is supported by the statement "[..] only one action accepts a given event occurrence, even if the event occurrence would satisfy multiple concurrently executing actions.", p. 250), 3) subthreads of a flow really execute in parallel, and 4) a signal can be sent before anyone is ready to receive it (this is the case as signals are stored in the objects associated with send/receive signal actions).

The two solutions presented by White [15] are not considered to be satisfactory. The first solution models the pattern by putting a verbal constraint on a couple of parallel activities stating that they are not to be run in parallel. The second solution provided by White uses the deferred choice pattern to capture the interleaved parallel routing pattern (as outlined in [3]). This solution suffers from combinatorial explosion as it boils down to enumerating all possible execution sequences of the activities involved.

3.8   Milestone

Description. A given activity can only be enabled if a certain milestone has been reached which has not yet expired. A milestone is defined as a point in the process where a given activity has finished and an activity following it has not yet started.

Solution in YAWL. YAWL directly supports the milestone pattern as it is based on Petri nets and therefore it can exploit the notion of state. A milestone can be realised through the use of arcs back and forth to a condition (which corresponds to the notion of place in Petri nets) testing whether a thread has reached a certain state (see Figure 9a).

Solution in UML. There is no direct support for the milestone in UML AD as the concept of state is not directly supported. A workaround can be devised with the use of signals, see Figure 9b. In the solution depicted in this figure, there is a race after the completion of A between continuing B, which has to await the receipt of $Signal1$ indicating that continuation of the thread is appropriate, and performing some other activity C. Activity C can only be performed after A has completed and before B has started. This is achieved by sending $Signal2$ which triggers $Signal3$ if indeed
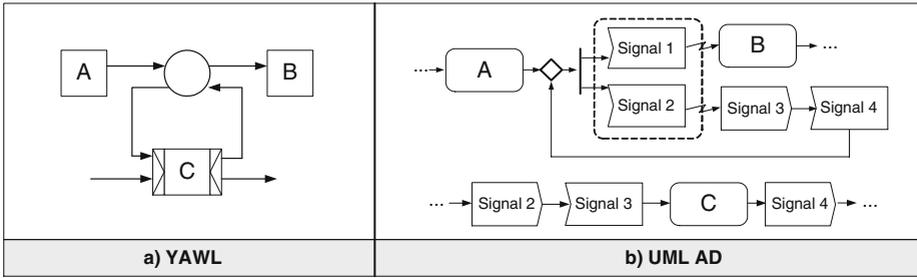
Fig. 9. Solutions for Milestone

the corresponding thread is in the correct state. If it is allowed to execute, then after completion, C issues $Signal4$ for indicating this.

The solution proposed by White [15] does not capture this pattern, as it does not model the expiration of the milestone. According to this solution an activity which *potentially* can be executed at a certain milestone, is *always* executed.

While workarounds exist for the state-based patterns, it is clear that mimicking the concept of a place as it exists in Petri nets through the use of signals may add a lot of complexity and could lead to models that are significantly less comprehensible. Moreover, many of the workarounds assume specific semantics for the constructs in UML AD. As yet, there are no formal semantics for UML AD and the workarounds may turn out to be invalid. Interpretations used by other authors suggest that there is currently no consensus on the semantics of the more advanced constructs.

3.9   Cancellation Patterns

There are two cancellation patterns: cancel activity and cancel case. As their semantics is straightforward we immediately focus on their solutions in YAWL and UML AD.

Solutions in YAWL. In Figure 10a execution of task B implies cancellation of task A, as this task is in the cancellation set of task B. In fact, any region can be chosen for cancellation so cancellation sets allow for cancellation of a single task, a whole case, and anything in between.

Solutions in UML. In UML AD, the cancel activity pattern can be captured as shown in Figure 10b. In this solution an interruptable region is used where apart from activity
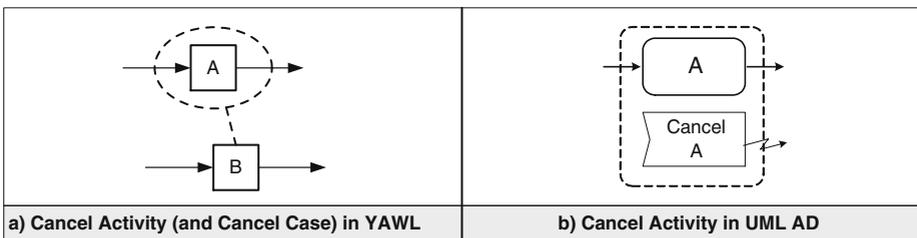

Fig. 10. Cancellation concepts

A there is an AcceptEventAction ready to accept a signal indicating that *A* should be cancelled. If such a signal is received during the execution of activity *A*, and as an interruptingEdge is used, everything in the region (in this case only activity *A*) will be cancelled (p. 409). The solution in Figure 10b is inspired by Figure 274 on p. 410 of the UML specification [9]. It is also identical to the solution presented by White [15]. Note that due to the statement "If an AcceptEventAction has no incoming edges, then the action starts when the containing activity or structured node [i.e. the interruptible region in this case] does..."(p. 334) no incoming edge is used for the cancellation event.

In UML AD, the cancel case pattern is captured by the ActivityFinalNode: "A token reaching an activity final node terminates the activity [..], it stops all executing actions in the activity, and destroys all tokens in object nodes, except in the output activity parameter nodes."(p. 357). White [15] offers two solutions: one along the lines of the approach to cancel activity (by making the process to be cancelled an activity and running it in parallel with the cancellation event) and another using ActivityFinalNode.

4   Conclusion

Table 1 summarises the evaluation in terms of the control-flow patterns. A '+' indicates *direct support* for the pattern (i.e. there is a construct in the language that directly supports the pattern). The evaluation of UML 2.0 is contrasted with a previous evaluation of UML 1.4[7]. Overall, UML 2.0 is a clear improvement over UML 1.4 in terms of *direct support* for the control-flow patterns. In regards to the patterns that UML 2.0 AD does not directly support we would like to make the following recommendations:

- Given the difficulties in supporting state-based patterns, most notably the Interleaved Parallel Routing pattern and the Milestone pattern, it may be worthwhile to provide direct support for the notion of place as it exists in Petri nets. Petri net places capture the notion of "waiting state" in a much less restrictive way than AcceptEventAction do. Similar to YAWL, one could then allow for implicit places thereby avoiding places that unnecessarily clutter up the diagram.
- UML AD currently does not support the creation of new instances of an activity while other instances of that activity are already running. This could be resolved through extensions to the ExpansionRegion construct along the lines of the "multiple instance" tasks in YAWL.
- Given the lack of support for the Synchronising Merge, a concept similar to the OR-join as it exists in YAWL could be added to UML AD.

During this pattern-based analysis, we have identified several ambiguities in the current UML specification, for example regarding the behaviour of expansion regions when the size(s) of the input and output collections do not match as mentioned in section 3.5, or the behaviour of signals that are raised before any accept signal can consume them (section 3.7). In general, such ambiguities can be resolved by identifying relevant passages in the specification and giving them an interpretation, as we have done in this paper, but a formalisation would help in making more precise and reliable interpretations. Unfortunately, the UML AD notation is not yet formalised (although work in this direction is

---

[7] This evaluation is based on [5] and the table presented at `www.workflowpatterns.com`.

Table 1. Comparison of UML AD version 2.0 and version 1.4

Nr	Pattern	2.0	1.4	Nr	Pattern	2.0	1.4
1	Sequence	+	+	11	Implicit Termination	+	−
2	Parallel Split	+	+	12	MI without Synchronization	+	−
3	Synchronisation	+	+	13	MI with a priori Design Time Knowledge	+	+
4	Exclusive Choice	+	+	14	MI with a priori Runtime Knowledge	+	+
5	Simple Merge	+	+	15	MI without a priori Runtime Knowledge	−	−
6	Multiple Choice	+	−	16	Deferred Choice	+	+
7	Synchronising Merge	−	−	17	Interleaved Parallel Routing	−	−
8	Multiple Merge	+	−	18	Milestone	−	−
9	Discriminator	+	−	19	Cancel Activity	+	+
10	Arbitrary Cycles	+	−	20	Cancel Case	+	+

ongoing e.g. [13]) and there are inherent difficulties in assessing a language that does not have a commonly agreed upon formal semantics nor an execution environment. We hope, however, that the analysis reported here, where different solutions are presented and discussed in both UML and a formalised language, namely YAWL, will serve to clarify (even if not directly formalise) the semantics of many of the language constructs, and thereby motivate further improvements to the language.

References

1. W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL System. In A. Persson and J. Stirna, editors, *Proc. of the 16th Int. Conf. on Advanced Information Systems Engineering (CAiSE'04)*, volume 3084 of *LNCS*, pages 142–159. Springer, 2004.
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. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
4. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual Modelling of Workflows. In M.P. Papazoglou, editor, *Proc. of the 14th Int. Object-Oriented and Entity-Relationship Modelling Conf. (OOER)*, volume 1021 of *LNCS*, pages 341–354. Springer, 1998.
5. M. Dumas and A. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In M. Gogolla and C. Kobryn, editors, *Proc. of the 4th Int. Conf. on the Unified Modeling Language (UML01)*, volume 2185 of *LNCS*, pages 76–90. Springer Verlag, 2001.
6. P. Green and M. Rosemann. Applying Ontologies to Business and Systems Modeling Techniques and Perspectives: Lessons Learned. *Journal of Database Management*, 15(2):105–117, 2004.
7. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
8. J-H. Kim and C. Huemer. Analysis, Transformation, and Improvements of ebXML Choreographies Based on Workflow Patterns. In R. Meersman and Z. Tari, editors, *Proc. of the OTM Confederated Int. Conf. (CoopIS, DOA, and ODBASE), Part I*, volume 3290 of *LNCS*, pages 66–84. Springer, 2004.
9. OMG. UML 2.0 Superstructure Specification. UML 2.0 Superstructure FTF convenience document ptc/04-10-02, 2004. www.omg.org/cgi-bin/doc?ptc/2004-10-02.

10. A.L. Opdahl and B. Henderson-Sellers. Ontological Evaluation of the UML Using the Bunge-Wand-Weber Model. *Software and System Modeling*, 1(1):43–67, 2002.
11. N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow Resouce Patterns: Identification, Representation and Tool Suppport. In O. Pastor and J. Falcão e Cunha, editors, *Proc. of 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE05)*, volume 3520 of *LNCS*, pages 216–232. Springer, 2005.
12. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Data Patterns. In L. Delcambre, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *to appear in Proc. of 24th Int. Conf. on Conceptual Modeling (ER05)*, LNCS. Springer Verlag, Oct 2005.
13. H. Störrle. Semantics of Control-Flow in UML 2.0 Activities. In P. Bottoni, C. Hundhausen, S. Levialdi, and G. Tortora, editors, *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04)*, pages 235–242. Springer Verlag, 2004.
14. WfMC. Workflow Management Coalition Terminology & Glossary, Document Number WFMC-TC-1011, Document Status - Issue 3.0. Technical report, Workflow Management Coalition, Brussels, Belgium, 1999.
15. S. White. Process Modeling Notations and Workflow Patterns. In L. Fischer, editor, *Workflow Handbook 2004*, pages 265–294. Future Strategies Inc., Lighthouse Point, FL, USA, 2004.
16. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In Il-Y. Song, S. W. Liddle, T. W. Ling, and P. Scheuermann, editors, *Proc. of 22nd Int. Conf. on Conceptual Modeling (ER 2003)*, volume 2813 of *LNCS*, pages 200–215. Springer, 2003.
17. P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and N. Russell. Pattern-based Analysis of UML Activity Diagrams. BETA Working Paper Series, WP 129, Eindhoven University of Technology, Eindhoven, The Netherlands, 2004. http://www.bpm.fit.qut.edu.au/projects/babel/docs/p242.pdf.
18. P. Wohed, E. Perjons, M. Dumas, and A.H.M. ter Hofstede. Pattern Based Analysis of EAI Languages - The Case of the Business Modeling Language. In *Proc. of the 5th Int. Conf. on Enterprise Information Systems (ICEIS 2003)*, volume 3, pages 174–184, 2003.
19. M.T. Wynn, D. Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-Join in Workflow Using Reset Nets. In G. Ciardo and P. Darondeau, editors, *Proc. of 26th Int. Conf. on Applications and Theory of Petri Nets 2005*, volume 3536 of *LNCS*, pages 423–443. Springer, 2005.