

# WORKFLOW CONTROL-FLOW PATTERNS

## A Revised View\*

Nick Russell<sup>1</sup>, Arthur H.M. ter Hofstede<sup>1</sup>,

<sup>1</sup>*BPM Group, Queensland University of Technology*

*GPO Box 2434, Brisbane QLD 4001, Australia*

{n.russell,a.terhofstede}@qut.edu.au

Wil M.P. van der Aalst<sup>1,2</sup>, Nataliya Mulyar<sup>2</sup>

<sup>2</sup>*Department of Technology Management, Eindhoven University of Technology*

*PO Box 513, NL-5600 MB Eindhoven, The Netherlands*

{w.m.p.v.d.aalst,n.mulyar}@tm.tue.nl

### Abstract

The *Workflow Patterns Initiative* was established with the aim of delineating the fundamental requirements that arise during business process modelling on a recurring basis and describe them in an imperative way. The first deliverable of this research project was a set of twenty patterns describing the control-flow perspective of workflow systems. Since their release, these patterns have been widely used by practitioners, vendors and academics alike in the selection, design and development of workflow systems [vdAtHKB03]. This paper presents the first systematic review of the original twenty control-flow patterns and provides a formal description of each of them in the form of a Coloured Petri-Net (CPN) model. It also identifies twenty three new patterns relevant to the control-flow perspective. Detailed context conditions and evaluation criteria are presented for each pattern and their implementation is assessed in fourteen commercial offerings including workflow and case handling systems, business process modelling formalisms and business process execution languages.

## 1 Introduction

It has been seven years since the *Workflow Patterns Initiative* first commenced identifying the core architectural constructs inherent in workflow technology [vdABtHK00, vdAtHKB03]. The original objective was to delineate the fundamental requirements that arise during business process modelling on a recurring basis and describe them in an imperative way. A patterns-based approach was taken to describing these requirements as it offered both a language-independent and technology-independent means

---

\*This research is conducted in the context of the *Patterns for Process-Aware Information Systems (P4PAIS)* project which is supported by the Netherlands Organisation for Scientific Research (NWO). It is also receives partial support from the Australian Research Council under the Discovery Grant *Expressiveness Comparison and Interchange Facilitation between Business Process Execution Languages*.

of expressing their core characteristics in a form that was sufficiently generic to allow for its application to a wide variety of offerings.

One of the drivers for this research was the observation that although many workflow systems were based on similar conceptual underpinnings, they differed markedly in their *expressive power* and the range of concepts that they were able to capture. Indeed these differences were so significant that it raised the question of how the *suitability* of specific tools for specific purposes might be evaluated. Despite the plethora of workflow systems in both the commercial and research domains, there was a notable absence of core foundational concepts that individual offerings could be expected to support or that could be used as a basis for comparison. This absence differs markedly from other areas of information systems such as database design or transaction management which are based on formal conceptual foundations that have effectively become de facto standards.

In line with the traditional patterns approaches employed by Alexander [AIS77] and the Gang of Four [GHJV95], that are based on broad survey of existing problems and practices within a particular domain, the original twenty control-flow patterns were identified through a comprehensive evaluation of workflow systems and process modelling formalisms. They describe a series of constructs that are embodied in existing offers in response to actual modelling requirements. The imperative approach employed in their description ensures that their intent and function is clearly presented without mandating a specific implementation approach. An overriding objective was that they describe control-flow characteristics which it would be *desirable* to support in a given offering.

The publication of the original patterns in 2000 [vdAtHKB03] had a galvanising effect on the workflow community. It provided clarity to concepts that were not previously well-defined and provided a basis for comparative discussion of the capabilities of individual workflow systems. Amongst some vendors, the extent of patterns support soon became a basis for product differentiation and promotion.

Although initially focussed on workflow systems, it soon became clear that the patterns were applicable in a much broader sense and they were used to examine the capabilities of business process modelling languages such as BPMN, UML Activity Diagrams and EPCs, web service composition languages such as WCSI and business process execution languages such as BPML, XPD and BPEL. Their coverage was also extended to the other classical workflow perspectives [JB96] and a series of new patterns were identified in the data [RtHEvdA05], resource [RvdAtHE05] and exception [RvdAtH06] perspectives. These patterns have been used to extend the reviews described above and have provided a series of new insights into the capabilities of particular offerings.

The control-flow patterns have been subject to ongoing scrutiny over the past few years particularly in terms of their completeness and precision. In order to ensure their continued relevance, a comprehensive review of the control-flow perspective has been conducted over the past six months, the results of which are presented in this document. As well as assessing their continued applicability, one of the major objectives of this review was to thoroughly overhaul the description of each of the patterns and to put them on a more formal footing to remove any potential ambiguities that may have previously existed. This has been achieved by augmenting each pattern description with a Coloured Petri-Net model [Jen97] describing its operation. There is also a more detailed discussion of the operational context in which each of the pat-

terns is intended to function and a specific set of evaluation criteria which describe how pattern support is assessed in an offering.

This review has confirmed the continued applicability of the original twenty patterns, all of which continue to be widely supported. It has also identified twenty three new patterns which are applicable to the control-flow perspective, some of which are based on refinements of the original patterns and many of which are the result of critical assessment of additional factors which are relevant to the control-flow perspective. A comprehensive review of patterns support has been conducted in fourteen distinct offerings including workflow systems (Staffware, WebSphere MQ, COSA, iPlanet, SAP Workflow and FileNet), case handling systems (FLOWer), business process modelling languages (BPMN, UML 2.0 Activity Diagrams and EPCs) and business process execution languages (BPEL4WS, WebSphere BPEL, Oracle BPEL and XPDL).

The remainder of this paper proceeds as follows: Section 2 provides background information on the patterns and discusses related work. Section 3 presents a completely revised set of descriptions for the original twenty patterns. Section 4 documents the new control-flow patterns that have been identified. Section 5 discusses the results of the product evaluations which have been conducted using the complete set of patterns. Section 6 discusses the relationships which exist between the patterns and Section 7 concludes the paper.

## 2 Background and Related Work

The notion of patterns as a means of categorising recurring problems and solutions in a particular domain is generally attributed to Christopher Alexander [AIS77] as is the concept of a patterns language for describing the inter-relationships between specific patterns. The original patterns work centred on the field of architecture, however the concept has general applicability and has been used widely in a number of other domains. It has had most impact however in the field of information technology where patterns have been used to categorise the major concepts in a number of areas including system design [GHJV95], business analysis [Hay95, Fow96], business process design [EP00], software architecture [BMR<sup>+</sup>96, SSRB00] and enterprise application integration [HW04].

The application of a patterns-based approach to the identification of generic workflow constructs was first proposed in [vdABtHK00], which identified several control-flow patterns relevant to the control-flow perspective of workflow systems. This work was subsequently expanded to encompass twenty control-flow patterns together with a formalisation and analysis of their implementation in fifteen commercial and research workflow systems.

This work triggered research efforts in two main directions: the first of these being the use of the patterns to establish a formal basis for understanding the requirements of the control-flow perspective, the second being the use of the patterns to evaluate the capabilities of a series of business process modelling languages and web services standards.

In [Kie03], the patterns were used to motivate an investigation into the fundamentals of workflow technology, in particular the expressiveness of various approaches to implementing control-flow constructs in workflow systems. In [DtH01],

they were first utilised to examine the capabilities of specific modelling languages, in this case UML 1.4 Activity Diagrams, in an effort to understand its strengths, weaknesses and also to suggest areas for possible improvement. This research strategy led to a series of subsequent investigations into the expressiveness of languages including BPEL4WS [WvdADtH03], BML [WPDtH03], UML 2.0 Activity Diagrams [WvdAD<sup>+</sup>05b, RvdAtHW06] and BPMN [Whi04, WvdAD<sup>+</sup>05a].

The original workflow patterns focussed on the control-flow perspective, however as noted in [JB96], a comprehensive description of a workflow process also requires consideration of the data and resource perspectives. With this requirement in mind, the *Workflow Patterns Initiative* was subsequently extended to include 40 patterns in the data perspective [RtHEvdA05] and 43 patterns in the resource perspective [RtHEvdA05, RvdAtHE05]. Subsequent research [RvdAtH06] has also investigated inter-relationships between these perspectives and has proposed a patterns-based approach to workflow exception handling. There has also been research into the use of patterns for characterising service interactions [BDtH05]. Most recently, the *Workflow Pattern Specification Language* (WPSL) [MvdAtHR06] has been developed. This is a formal language motivated by the workflow patterns which provides a deterministic basis for capturing and comparing the control-flow capabilities of Process-Aware Information Systems (PAISs).

The workflow patterns have provided the conceptual basis for the YAWL workflow system [vdAtH05]. YAWL is an acronym for “Yet Another Workflow Language”, an initiative that aims to provide both a workflow language that is based on the workflow patterns and also an open-source reference implementation that demonstrates the manner in which these constructs can interoperate.

Since their release, the workflow patterns have been used for a wide variety of purposes including evaluation of PAISs, tool selection, process design, education and training. They have been enthusiastically received by both industry practitioners and academics alike. The original *Workflow Patterns* paper has been cited by over 150 academic publications and the workflow patterns website has been visited more than 40,000 times. Full details can be found at <http://www.workflowpatterns.com>. The webpage also contains an “impact page” describing the tangible effect of the patterns. Many organizations around the globe and in particular in The Netherlands have used the patterns for workflow management system selection. The patterns have also directly influenced the development of many academic and commercial tools and have influenced the development of standards including BPMN and BPEL.

The contribution of this paper to the problem domain is twofold: (1) it provides a critical evaluation of the existing patterns on a formal basis using CPN Tools and (2) it defines 23 new patterns whose semantics are also formalised and expressed in terms of CPN Tools.

### 3 A Review of the Original Control-Flow Patterns

In this section, we examine the original twenty control-flow patterns and revise their definitions. The format in which each pattern is presented has been significantly expanded to incorporate a more detailed description of the pattern and a more comprehensive rationale for its usage.

One of the major areas of ambiguity in regard to the original descriptions related

to varying interpretations of their applicability and operation. In an effort to remove this area of uncertainty, we provide a detailed definition of the operational semantics of each pattern in the form of a Coloured Petri-Net (CPN) diagram<sup>1</sup> [Jen97] together with the context conditions that apply to the pattern.

Once again, we examine the operational support for the pattern across a series of contemporary offerings in the business process management field and we extend the range of products we review to include not only current commercial workflow and case handling products, but also business process modelling notations such as BPMN, EPCs and UML 2.0 Activity Diagrams and business process execution languages such as XPDL and BPEL. A significant change in the manner in which this process is undertaken is the establishment of a definitive set of evaluation criteria for rating patterns support in a given offering. Details of these criteria are included for each pattern.

### 3.1 Review and Augmentation of the Original Control-Flow Patterns

The original set of twenty patterns in the control-flow perspective were derived from a detailed examination of contemporary workflow systems and business process modelling notations in order to identify *generic, recurring constructs*. They have proven to be extremely popular with both theoreticians and practitioners alike as they provide a useful basis for discussing and comparing the capabilities and expressiveness of individual offerings in a manner which is independent of specific modelling formalisms and implementation technologies.

More recently, it has become clear that there are a number of additional scenarios in the control-flow perspective that require categorization. Moreover, there are several of the original patterns that would benefit from a more precise description in order to remove potential ambiguities in relation to the concepts that they are intended to represent. Indeed with a more rigorous foundation, it becomes possible to further refine several of the patterns into forms that more effectively describe and distinguish between the distinct situations to which they might be applicable. As a consequence of this reflection there have been some changes to the existing pattern definitions and the identification of twenty three new patterns, some of them new and some based on specializations of existing patterns.

The original *Synchronizing Merge* pattern did not adequately differentiate between possible context assumptions, each of which has a distinct semantics. Consequently, it has now been divided into three distinct patterns:

- the *Structured Synchronizing Merge* (WCP-7), which restricts the original pattern to use in a structured workflow context and sees it take the form of a join which is paired with a specific preceding Multi-Choice i.e. there is a one-to-one correspondence between split and join;
- the *Acyclic Synchronizing Merge* (WCP-37), which recognizes tractable OR-join implementations, based on conventions such as true/false token passing, which allow their evaluation to be based on information directly available to the merge (i.e. local semantics) but which are unable to deal with loops; and

---

<sup>1</sup>We use CPN Tools for preparation and validation of these models, copies of which are available from <http://www.workflowpatterns.com>. More details on CPN Tools can be found at <http://wiki.daimi.au.dk/cpntools/>

- the *General Synchronizing Merge* (WCP-38), which denotes a general solution to OR-join evaluation based on a thorough analysis of the current and potential future states of a process instance (i.e. non-local semantics).

In a similar vein, the original *Discriminator* pattern did not differentiate between distinct implementation approaches and the degree to which they were able to deal with concurrency within a process instance. It is now recognized as having three distinct forms:

- the *Structured Discriminator* (WCP-9), where it operates in a safe and structured workflow context (i.e. it is assumed that each branch executes precisely once before a reset takes place and there is a one-to-one correspondence between splits and joins);
- the *Blocking Discriminator* (WCP-28), where concurrency within a process instance is dealt with by blocking additional execution threads within a given branch until the discriminator has reset; and
- the *Cancelling Discriminator* (WCP-29), where remaining incoming branches which are still executing after the discriminator fires are cancelled.

The *Partial (or N-out-of-M) Join* pattern, which had previously only been denoted as a sub-case of the *Discriminator*, is now also recognized as a pattern in its own right with three distinct forms: the *Structured Partial Join* (WCP-30), the *Blocking Partial Join* (WCP-31) and the *Cancelling Partial Join* (WCP-32).

The *Structured Loop* (WCP-21) pattern has been introduced to deal with more restrictive forms of iteration such as while and repeat loops which are not adequately covered by the *Arbitrary Cycles* pattern. Similarly the *Recursion* pattern (WCP-22) covers repetitive activity execution which is based on self-invocation.

Existing multiple instance patterns assume that all activity instances must complete before a subsequent activity can be enabled. In recognition that more efficient means exist of implementing concurrency with respect to overall process execution and determination as to when the process can proceed beyond the multiple instance task, three new patterns have been introduced: the *Static Partial Join for Multiple Instances* (WCP-34), the *Cancelling Partial Join for Multiple Instances* (WCP-35) and the *Dynamic Partial Join for Multiple Instances* (WCP-36).

The ability to respond to external signals within a process instance was not well covered by the original patterns other than by the *Deferred Choice* (WCP-16) which allows a decision regarding possible execution paths to be based on environmental input. To remedy this, two new patterns are introduced to denote the ability of external signals to affect process execution. These are the *Transient Trigger* (WCP-23) and the *Persistent Trigger* (WCP-24).

The *Interleaved Parallel Routing* pattern (WCP-17) is extended to two new patterns:

- the *Critical Section* pattern (WCP-39), which provides the ability to prevent concurrent execution of specific parts of a process; and
- the *Interleaved Routing* pattern (WCP-40), which denotes situations where a group of activities can be executed sequentially in *any* order.

Previous notions of cancellation only related to individual activities and complete process instances (cases). In order to deal with cancellation in a more general sense,

the *Cancel Region* pattern (WCP-25) has been introduced, which allows for arbitrary groups of activities in a process to be cancelled during execution. Similarly, in recognition that the semantics of cancelling a multiple instance activity is different to those associated with cancelling a normal activity, the *Cancel Multiple Instance Activity* (WCP-26) pattern has also been included and there is also a *Complete Multiple Instance Activity* (WCP-27) to handle the situation where a multiple instance activity is forced to complete during execution.

Other new inclusions are the *Generalised AND-Join* pattern (WCP-33) which defines a model of AND-join operation for use in highly concurrent processes, *Thread Merge* (WCP-41) and *Thread Split* (WCP-42) which provide for coalescence and divergence of distinct threads of control along a single branch and *Explicit Termination* (WCP-43) which provides an alternative approach to defining process completion.

### 3.2 Context Assumptions

As far as possible, each pattern is illustrated using the Coloured Petri-Net (CPN) formalism. This allows us to provide a precise description of each pattern that is both *deterministic* and *executable*. This approach becomes increasingly important with some of the revised pattern definitions (as well as for some of the new patterns) as the actual context in which the pattern can occur requires detailed description.

There are some blanket assumptions that apply to all of the CPN models used in this paper. For each of them, we adopt a notation in which input places are labelled  $i1 \dots in$ , output places are labelled  $o1 \dots on$ , internal places are labelled  $p1 \dots pn$  and transitions are labelled  $A \dots Z$ . In the case where either places or transitions serve a more significant role in the context of the pattern, they are given more meaningful names (e.g. buffer or anti-place). In general, transitions are intended to represent tasks or activities in processes and places are the preceding and subsequent states which describe when the activity can be enabled and what the consequences of its completion are.

Unless stated otherwise, we assume that the tokens flowing through a CPN model that signify control-flow are typed CID (short for “Case ID”) and that each executing case (i.e. process instance) has a distinct case identifier. For most patterns, the assumption is also made that the model is *safe* i.e. that each place in the model can only contain at most one token (i.e. one thread of control for each case currently being executed). This provides clarity in regard to the way in which each of the CPN models describing pattern operation are intended to function.

Safe behaviour is not a mandatory quality of workflow systems. Some of the systems that we examine during the course of this paper, do implement safe process models whilst others do not. Where a system does provide a safe execution environment, this is typically achieved in one of two ways: either (1) during execution, the state of a given case is never allowed to transition into an unsafe state. This is the approach adopted by COSA, which blocks an activity’s execution where it has a token in the place immediately after it and allowing it to execute could potentially result in an unsafe state (i.e. the following place having two tokens in it). The other alternative (2) is to detect any unsafe situations that may arise and migrate them to safe states. An example of this is the strategy employed by Staffware (often referred to as the “Pac-Man approach”) where any additional triggerings received by an activity that is currently executing are coalesced into the same thread of control

resulting in a single thread of control being delivered to outgoing branches when the activity completes.

These variations in the ways in which distinct offerings implement concurrency with a process instance lead to differences in the ranges of patterns that they are able to support and the means by which they realise them. In the following sections, we will provide precise description of each pattern and examine the approaches to their implementation taken by various workflow tools and business process modelling languages.

### 3.3 Revisiting the Original Patterns

This section presents a revised description of the original twenty control-flow patterns previously presented in [vdAtHKB03]. Although this material is motivated by earlier research conducted as part of the *Workflow Patterns Initiative*<sup>2</sup>, the descriptions for each of these patterns have been thoroughly revised and a new set of evaluations have been undertaken. In several cases, detailed review of a pattern has indicated that there are potentially several distinct ways in which the original pattern could be interpreted and implemented. In order to resolve these ambiguities, we have taken the decision to base the revised definition of the original pattern on the *most restrictive* interpretation of its operation and to delineate this from other possible interpretations that could be made. In several situations, a substantive case exists for consideration of these alternative operational scenarios and where this applies, these are presented in the form of new control-flow patterns in Section 4.

#### 3.3.1 Basic Control-Flow Patterns

This class of patterns captures elementary aspects of process control and are similar to the definitions of these concepts initially proposed by the Workflow Management Coalition (WfMC) [Wor99].

##### Pattern WCP-1 (Sequence)

**Description** An activity in a workflow process is enabled after the completion of a preceding activity in the same process.

**Synonyms** Sequential routing, serial routing.

##### Examples

- The *verify-account* activity executes after the credit card details have been captured.
- The *codacil-signature* activity follows the *contract-signature* activity.
- A receipt is printed after the train ticket is issued.

**Motivation** The *Sequence* pattern serves as the fundamental building block for workflow processes. It is used to construct a series of consecutive activities which execute in turn one after the other. Two activities form part of a *Sequence* if there is a control-flow edge from one of them to the next which has no guards or conditions associated with it.

**Context** Figure 1 illustrates the *Sequence* pattern using the Coloured Petri-Net (CPN) formalism.

---

<sup>2</sup>See <http://www.workflowpatterns.com> for further details.



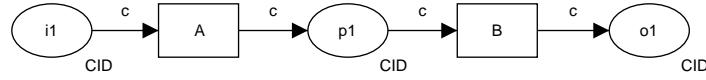


Figure 1: Sequence pattern

**Implementation** The *Sequence* pattern is widely supported and all of the workflow systems and business process modelling languages examined directly implement it.

**Issues** Although all of the offerings examined implement the *Sequence* pattern, there are however, subtle variations in the manner in which it is supported. In the main, these differences centre on how individual offerings deal with concurrency within a given process instance and also between distinct process instances. In essence these variations are characterised by whether the offering implements a safe process model or not. In CPN terms, this corresponds to whether each of the places in the process model such as that in Figure 1 are *1-bounded* (i.e. can only contain at most one token for a case) or not.

**Solutions** This issue is handled in a variety of differing ways. BPMN, XPD and UML 2.0 Activity Diagrams assume the use of a “token-based” approach to managing process instances and distinguishing between them, although no details are given as to how this actually occurs. Further, although individual tokens are assumed to be conserved during execution of a process instance, it is possible for an activity, split or join construct to specifically add or remove tokens during execution beyond what would reasonably be expected. Staffware simply ignores the issue and where a step receives two threads (or more) of execution at the same time, they are simply coalesced into a single firing of the step (thus resulting in race conditions). COSA adopts a prevention strategy, both by implementing a safe process model and also by disabling the activity(s) preceding a currently enabled activity and not allowing the preceding activity(s) to fire until the subsequent activity has completed.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which is able to provide a means of specifying the execution sequence of two (or more) activities. This may be based on directed arcs between activities or rules specifying the overall execution sequence.

### Pattern WCP-2 (Parallel Split)

**Description** The divergence of a branch into two or more parallel branches each of which execute concurrently.

**Synonyms** AND-split, parallel routing, parallel split, fork.

#### Examples

- After completion of the *capture enrolment* activity, run the *create student profile* and *issue enrolment confirmation* activities simultaneously.
- When an *intrusion alarm* is received, trigger the *despatch patrol* activity and the *inform police* activity immediately.
- Once the customer has paid for the goods, issue a receipt and pack them for despatch.

**Motivation** The *Parallel Split* pattern allows a single thread of execution to be split into two or more branches which can execute activities concurrently. These branches may or may not be re-synchronized at some future time.

**Context** Figure 2 illustrates the implementation of the *Parallel Split*. After activity A has completed, two distinct threads of execution are initiated and activities B and C can proceed concurrently.

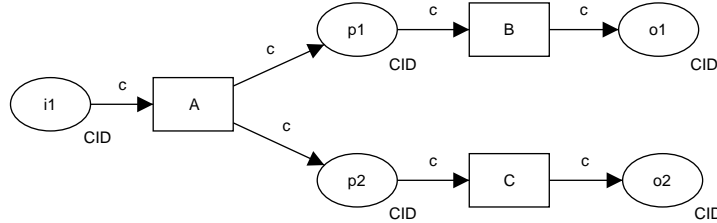


Figure 2: Parallel split pattern

**Implementation** The *Parallel Split* pattern is implemented by all of the offerings examined. It may be depicted either *explicitly* or *implicitly* in process models. Where it is represented explicitly, a specific construct exists for the *Parallel Split* with one incoming edge and two or more outgoing edges. Where it is represented implicitly, this can be done in one of two ways: either (1) the edge representing control-flow can split into two (or more) distinct branches or (2) the activity after which the *Parallel Split* occurs has multiple outgoing edges which do not have any conditions associated with them.

Of the offerings examined, Staffware, WebSphere MQ, FLOWer, COSA and iPlanet represent the pattern implicitly. SAP Workflow, EPCs and BPEL<sup>3</sup> do so with explicit branching constructs. UML 2.0 ADs, BPMN and XPDL allow it to be represented in both ways.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that provides a construct (either implicit or explicit) that allows the thread of control at a given point in a process model to be split into two or more concurrent branches.

### Pattern WCP-3 (Synchronization)

**Description** The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.

**Synonyms** AND-join, rendezvous, synchronizer.

### Examples

- The *despatch-goods* activity runs immediately after both the *check-invoice* and *produce-invoice* activities are completed.

<sup>3</sup>In general, the two BPEL implementations examined – WebSphere BPEL (which is part of WebSphere Process Server) and Oracle BPEL – provide a relatively faithful implementation of the BPEL 1.1 specification hence the evaluation results are identical for all three offerings. For this reason in this paper we do not list them individually unless there is a variation between them.

- *Cash-drawer reconciliation* can only occur when the store has been closed and the credit card summary has been printed.

**Motivation** *Synchronization* provides a means of reconverging the execution threads of two or more parallel branches. In general, these branches are created from a parallel split (AND-split) construct earlier in the process model. The thread of control is passed to the activity immediately following the synchronizer once all of the incoming branches have completed.

**Context** The behaviour of the *Synchronization* pattern is illustrated by the CPN model in Figure 3. There are two important context conditions associated with this pattern: (1) each incoming branch executes precisely once for a given case and (2) the synchronizer can only be reset (and fire again) once each incoming branch has completed. These conditions are important since if all incoming branches do not complete, then the synchronizer will deadlock and if more than one trigger is received on a branch, then the behaviour of the construct is undefined. They also serve to alleviate the concern as to whether all of the threads being synchronized relate to the same process instance. This issue becomes a significant problem in joins that do not have these restrictions.

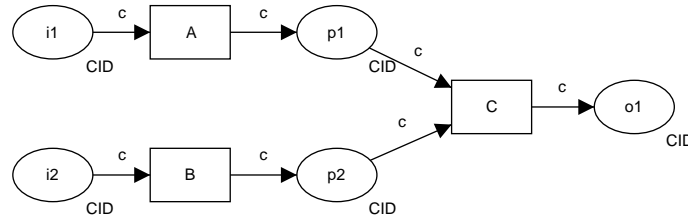


Figure 3: Synchronization pattern

**Implementation** Similar to the *Parallel Split* pattern, the synchronizer can either be represented *explicitly* or *implicitly* in a process model. Staffware has an explicit AND-join construct as do SAP Workflow, EPCs, BPMN and XPD. Other offerings – WebSphere MQ, FLOWer, COSA, iPlanet and BPEL – represent this pattern implicitly through multiple incoming (and unconditional) control edges to an activity. Only when each of these arcs has received the thread of control can the activity be enabled. UML 2.0 ADs allow it to be represented in both ways.

**Issues** The use of the *Synchronization* pattern can potentially give rise to a deadlock in the situation where one of the incoming branches fails to deliver a thread of control to the join construct. This could be a consequence of one of the activities in the branch failing to complete successfully (e.g. as a consequence of it experiencing some form of exception) or because the thread of control is passed outside of the branch.

**Solutions** None of the workflow systems or business process execution languages examined provide support for resolving this issue where the problem is caused by activity failure in one of the incoming branches however the structured nature of this pattern generally ensures that the second possible cause of deadlock does not arise.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct to merge several distinct threads of execution in different branches into a single thread of execution in a single branch. The merge occurs when a thread of control has been received on each of the incoming branches.

#### Pattern WCP-4 (Exclusive Choice)

**Description** The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on the outcome of a logical expression associated with the branch.

**Synonyms** XOR-split, exclusive OR-split, conditional routing, switch, decision, case statement.

#### Examples

- Depending on the volume of earth to be moved, either the *dispatch-backhoe*, *dispatch-bobcat* or *dispatch-D9-excavator* activity is initiated to complete the job.
- After the *review election* activity is complete, either the *declare results* or the *recount votes* activity is undertaken.

**Motivation** The *Exclusive Choice* pattern allows the thread of control to be directed to a specific activity depending on the outcome of a preceding activity, the values of elements of specific data elements in the workflow or the results of a user decision. The routing decision is made dynamically allowing it to be deferred to the latest possible moment at runtime.

**Context** The behaviour of the *Exclusive Choice* pattern is illustrated by the CPN model in Figure 4. Depending on the results of the **cond** expression, the thread of control is either routed to activity B or C. There are two context conditions associated with this pattern: (1) the information required to calculate the logical conditions on each of the outgoing branches must be available at runtime at the point at which the choice construct is reached in the process and (2) the condition associated with precisely one outgoing branch of the exclusive choice construct must evaluate to true.

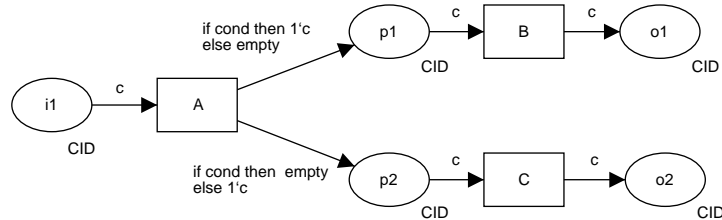


Figure 4: Exclusive choice pattern

**Implementation** Similar to the *Parallel Split* and *Synchronization* patterns, the *Exclusive Choice* pattern can either be represented explicitly via a specific construct or implicitly via disjoint conditions on the outgoing control-flow edges of an activity. Staffware, SAP Workflow, XPDL, EPCs and BPMN provide explicit XOR-split constructs. In the case of Staffware, it is a binary construct where as other offerings support multiple outgoing arcs. BPMN and XPDL provide for multiple outgoing edges as well as a default arc. Each edge has a condition associated with it and there is also the potential for defining the evaluation sequence but only one can evaluate to true at runtime. There is no provision for managing the situation where no default is specified and none of the branch conditions evaluate to true nor where more than one branch condition evaluates to true (simultaneously) and no evaluation sequence is specified. SAP Workflow provides three distinct means of implementing

this pattern: (1) based on the evaluation of boolean expression one of two possible branches chosen, (2) one of multiple possible branches is chosen based on the value of a specific data element (each branch has a nominated set of values which allow it to be selected and each possible value is assigned to exactly one branch) and (3) based on the outcome of a preceding activity, a specific branch is chosen (a unique branch is associated with each possible outcome). UML 2.0 ADs also provide a dedicated split construct although it is left to the auspices of the designer to ensure that the conditions on outgoing edges are disjoint (e.g. the same construct can be used for OR-splits as well). Likewise EPCs support the pattern in a similar fashion. The other offerings examined – WebSphere MQ, FLOWer, COSA, iPlanet and BPEL – represent the pattern implicitly, typically via conditions on the outgoing control-flow edges from an activity which must be specified in such a way that they are disjoint.

**Issues** One of the difficulties associated with this pattern is ensuring that precisely one outgoing arc is triggered when the *Exclusive Choice* is enabled.

**Solutions** The inclusion of default outgoing arcs on XOR-split constructs is an increasingly common means of ensuring that an outgoing arc is triggered (and hence the thread of control continues in the branch) when the XOR-split is enabled and none of the conditions on outgoing branches evaluate to true. An associated issue is ensuring that no more than one branch is triggered. There are two possible approaches to dealing with this issue where more than one of the arc conditions will potentially evaluate to true. The first of these is to simply select one of these arcs and allow it to proceed whilst ensuring that none of the other outgoing arcs are enabled. The second option, which is more practical in form, is to assign an evaluation sequence to the outgoing arcs which defines the order in which arc conditions will be evaluated. The means of determining which arc is triggered then becomes one of evaluating the arc conditions in sequential order until one evaluates to true. The arc is then triggered and the evaluation stops (i.e. no further arcs are triggered). In the event that none evaluate to true, then the default arc is triggered.

**Evaluation Criteria** Full support for this pattern is evidenced by an offering which provides a construct which enables the thread of control to be directed to exactly one of several outgoing branches. The decision as to which branch is selected is made at runtime on the basis of specific conditions associated with each of the branches.

### Pattern WCP-5 (Simple Merge)

**Description** The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

**Synonyms** XOR-join, exclusive OR-join, asynchronous join, merge.

#### Examples

- At the conclusion of either the *bobcat-excavation* or the *D9-excavation* activities, an estimate of the amount of earth moved is made for billing purposes.
- After the *cash-payment* or *provide-credit* activities, initiate the *produce-receipt* activity.

**Motivation** The *Simple Merge* pattern provides a means of merging two or more distinct branches without synchronizing them. As such, this presents the opportunity to simplify a process model by removing the need to explicitly replicate a sequence

of activities that is common to two or more branches. Instead, these branches can be joined with a simple merge construct and the common set of activities need only to be depicted once in the process model.

**Context** Figure 5 illustrates the behaviour of this pattern. Immediately after either activity A or B is completed, activity C will be enabled. There is no consideration of synchronization and it is a context condition of this pattern that the place at which the merge occurs (i.e. place p1 in Figure 5) is safe and can never contain more than one token.

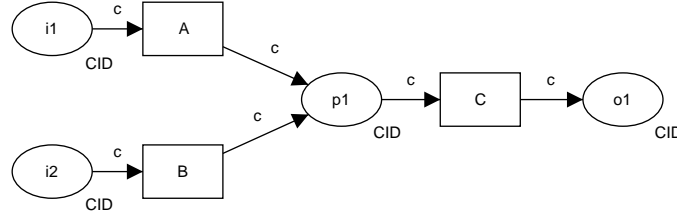


Figure 5: Simple merge pattern

**Implementation** Similar to patterns WCP2–WCP4 described above, this pattern can either be represented explicitly or implicitly. Staffware, SAP Workflow and UML 2.0 ADs provide specific join constructs for this purpose where as it is represented implicitly in WebSphere MQ, FLOWer, COSA and BPEL. BPMN and XPD L allow it to be represented in both ways.

**Issues** One issue that can arise with the use of this pattern occurs where it cannot be certain that the incoming place to the merge (p1) is safe.

**Solutions** In this situation, the context conditions for the pattern are not met and it cannot be used, however there is an alternative pattern – the *Multi-Merge* (WCP-8) – that is able to deal with the merging of branches in potentially unsafe process instances.

**Evaluation Criteria** Full support for this pattern in an offering is evidenced by the ability to merge several distinct branches into one such that each thread of control received on any incoming branch is immediately passed onto the outgoing branch.

### 3.3.2 Advanced Branching and Synchronization Patterns

This section presents a series of patterns which characterise more complex branching and merging concepts which arise in business processes. Although relatively commonplace in practice, these patterns are often not directly supported or even able to be represented in many commercial offerings. The original control-flow patterns identified four of these patterns: *Multi-Choice*, *Synchronizing Merge*, *Multi-Merge* and *Discriminator*.

In this revision, the *Multi-Choice* and *Multi-Merge* have been retained in their previous form albeit with a more formal description of their operational semantics. For the other patterns however, it has been recognized that there are a number of distinct alternatives to the manner in which they can operate. The original *Synchronizing Merge* now provides the basis for three patterns: the *Structured Synchronizing Merge* (WCP-7), the *Acyclic Synchronizing Merge* (WCP-37) and the *General Synchronizing Merge* (WCP-38).

In a similar vein, the original *Discriminator* pattern is divided into six distinct patterns: the *Structured Discriminator* (WCP-9), the *Blocking Discriminator* (WCP-28), the *Cancelling Discriminator* (WCP-29), the *Structured Partial Join* (WCP-30), the *Blocking Partial Join* (WCP-31) and the *Cancelling Partial Join* (WCP-32). One other addition has been the *Generalized AND-Join* (WCP-33) which identifies a more flexible AND-join variant useful in concurrent processes.

Of these patterns, the original descriptions for the *Synchronizing Merge* and the *Discriminator* are superseded by their structured definitions and are described in detail in this section. The remaining new patterns are presented in Section 4.

### Pattern WCP-6 (Multi-Choice)

**Description** The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is passed to one or more of the outgoing branches based on the outcome of distinct logical expressions associated with each of the branches.

**Synonyms** Conditional routing, selection, OR-split, multiple choice.

#### Example

- Depending on the nature of the emergency call, one or more of the *despatch-police*, *despatch-fire-engine* and *despatch-ambulance* activities is immediately initiated.

**Motivation** The *Multi-Choice* pattern provides the ability for the thread of execution to be diverged into several concurrent threads on a selective basis. The decision as to whether to start a given thread is made at run-time on the basis of workflow control data and execution information. This pattern is essentially an analogue of the *Exclusive Choice* pattern (WCP-4) in which the conditions on the outgoing branches are not required to be disjoint.

**Context** The operation of the *Multi-Choice* pattern is illustrated in Figure 6. After activity A has been triggered, the thread of control can be passed to one or both of the following branches depending on the evaluation of the conditions associated with each of them. The main context criterion for this pattern is that the information required to calculate the logical conditions on each of the outgoing branches is available at runtime at the point at which the *Multi-Choice* construct is reached in the process.

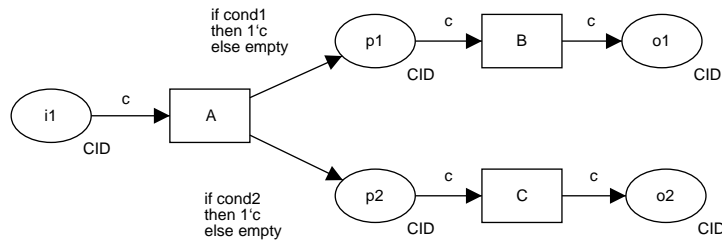


Figure 6: Multi-choice pattern

**Implementation** As with other branching and merging constructs, the *Multi-Choice* pattern can either be represented implicitly or explicitly. WebSphere MQ captures it implicitly via (non-disjoint) conditions on outgoing arcs from a process or block construct, COSA and iPlanet do much the same via overlapping conditions on outgoing arcs from activities and outgoing routers respectively. Both COSA and iPlanet

allow for relatively complex expressions to be specified for these outgoing branches and iPlanet also allows for procedural elements to form part of these conditions. The modelling and business process execution languages examined tend to favour the use of explicit constructs for representing the pattern: BPEL via conditional links within the <flow> construct, UML 2.0 ADs via the ForkNode with guards conditions on the outgoing arcs and EPCs via textual notations to the OR-split construct. BPMN and XPDL provide three alternative representations including the use of an implicit split with conditions on the arcs, an OR-split or a complex gateway.

**Issues** Two issues have been identified with the use of this pattern. First, as with the *Exclusive Choice*, an issue that also arises with the use of this pattern is ensuring that at least one outgoing branch is selected from the various options available. If this is not the case, then there is the potential for the workflow to deadlock. Second, where an offering does not support the *Multi-Choice* construct directly, the question arises as to whether there are any indirect means of achieving the same behaviour.

**Solutions** With respect to the first issue, the general solution to this issue is to enforce the use of a default outgoing arc from a *Multi-Choice* construct which is enabled if none of the conditions on the other outgoing arcs evaluate to true at runtime. For the second issue, a work-around that can be used to support the pattern in most offerings is based on the use of an AND-split immediately followed by an (binary) XOR-split in each subsequent branch. Another is the use of an XOR-split with an outgoing branch for each possible activity combination, e.g. a *Multi-Choice* construct with outgoing branches to activities A and B would be modelled using an XOR-split with three outgoing branches – one to activity A, another to activity B and a third to an AND-split which then triggered *both* activities A and B. Further details on these transformations can be found in [vdAtHKB03].

**Evaluation Criteria** Full support for this pattern is evidenced by the availability of a construct which allows the thread of control to be diverged into one or more branches on the basis of conditions associated with each of the branches. Note that the work-around based on XOR-splits and AND-splits is not considered to constitute support for this pattern.

---

The first six patterns focus primarily on workflow structure and essentially correspond to specific constructs that can be expected to appear in a workflow language. Indeed the first five patterns are directly supported in all of the offerings examined and the majority of them also support the sixth as well. In each of these cases, the CPN model presented to illustrate the pattern corresponds very closely to the actual realisation of the pattern in individual offerings.

The remainder of the patterns that we will describe have a distinct focus which centres on their actual behaviour in a workflow context. As with the first six patterns, their operation is also described in terms of a CPN model, however the emphasis is on the actual semantics of the pattern being presented rather than the way in which it is realised. As a consequence there is not such a close structural correspondence between the CPN models for individual patterns and the form in which they are realised in individual offerings and the direct replication of the CPN model in a specific workflow language does not necessarily demonstrate support for the pattern unless the language also replicates the semantics associated with the pattern.

---



### Pattern WCP-7 (Structured Synchronizing Merge)

**Description** The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled.

**Synonyms** Synchronizing join, synchronizer.

#### Example

- Depending on the type of emergency, either or both of the *despatch-police* and *despatch-ambulance* activities are initiated simultaneously. When all emergency vehicles arrive at the accident, the *transfer-patient* activity commences.

**Motivation** The *Synchronizing Merge* pattern provides a means of merging the branches resulting from a specific Multi-Choice or OR-split construct earlier in a workflow process into a single branch. Implicit in this merging is the synchronization of all of the threads of execution resulting from the preceding Multi-Choice.

**Context** As already indicated, the *Synchronizing Merge* construct provides a means of merging the branches from a preceding *Multi-Choice* construct and synchronizing the threads of control flowing along each of them. It is not necessary that all of the incoming branches to the *Synchronizing Merge* are active in order for the construct to be enabled, however all of the threads of control associated with the incoming branches must have reached the Synchronizing Merge before it can fire. As such there are four context conditions associated with the use of this pattern:

1. There must be a single *Multi-Choice* construct earlier in the process model with which the *Synchronizing Merge* is associated and it must merge all of the branches emanating from the *Multi-Choice*. These branches must either flow from the *Multi-Choice* to the *Synchronizing Merge* without any splits or joins or they must be structured in form (i.e. balanced splits and joins) such that it is not possible for the *Synchronizing Merge* to receive multiple triggers on the same branch once the *Multi-Choice* has been enabled;
2. The *Multi-Choice* construct must not be re-enabled before the associated *Synchronizing Merge* construct has fired;
3. Once the *Multi-Choice* has been enabled none of the activities in the branches leading to the *Synchronizing Merge* can be cancelled before the merge has been triggered. The only exception to this is that it is possible for *all* of the activities leading up to the *Synchronizing Merge* to be cancelled; and
4. The *Synchronizing Merge* must be able to resolve the decision as to when it should fire based on local information available to it during the course of execution. Critical to this decision is knowledge of how many branches emanating from the *Multi-Choice* are active and require synchronization. This is crucial in order to remove any potential for the “vicious circle paradox” [Kin06] to arise where the determination of exactly when the merge can fire is based on non-local semantics which by necessity include a self-referencing definition and make the firing decision inherently ambiguous.

**Implementation** Addressing the last of the context conditions without introducing non-local semantics for the *Synchronizing Merge* can be achieved in several ways including (1) structuring of the process model following a *Multi-Choice* such that the subsequent *Synchronizing Merge* will always receive precisely one trigger on each of its incoming branches and no additional knowledge is required to make the decision as to when it should be enabled, (2) by providing the merge construct with knowledge of how many incoming branches require synchronization and (3) by undertaking a thorough analysis of possible future execution states to determine when the *Synchronizing Merge* can fire.

The first of these implementation alternatives forms the basis for this pattern and is illustrated in Figure 7. It involves adding an alternate “bypass” path around each branch from the multi-merge to the *Synchronizing Merge* which is enabled in the event that the normal path is not chosen. The “bypass” path is merged with the normal path for each branch prior to the *Synchronizing Merge* construct ensuring that it always gets a trigger on all incoming branches and can hence be implemented as an AND-join construct.

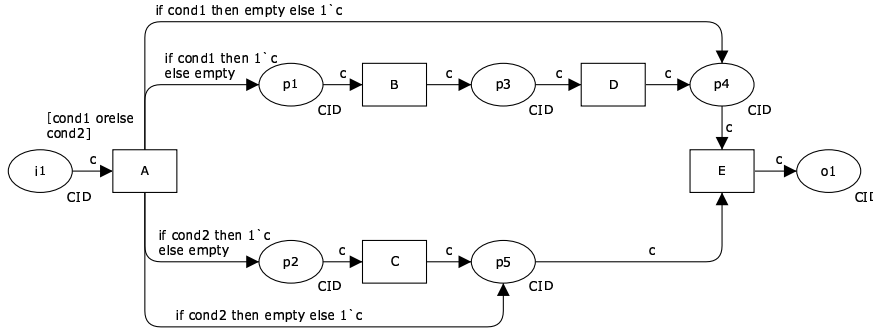


Figure 7: Structured synchronizing merge pattern

The second implementation alternative forms the basis for the *Acyclic Synchronizing Merge* (WCP-37) pattern. It can be facilitated in several distinct ways. One option [Rit99] is based on the immediate communication from the preceding *Multi-Choice* to the *Synchronizing Merge* of how many branches require synchronization. Another option (illustrated in Figure 8) involves the introduction of true/false tokens following a multi-merge indicating whether a given branch has been chosen or not. This pattern variant is discussed further on page 69.

The third implementation alternative – undertaking a complete execution analysis to determine when the *Synchronizing Merge* should be enabled – forms the basis for the *General Synchronizing Merge* (WCP-38) pattern and is discussed on page 71.

The *Structured Synchronizing Merge* can be implemented in any workflow language which supports the *Multi-Choice* construct and can satisfy the four context conditions listed above. It is directly supported in WebSphere MQ, FLOWer, FileNet, BPMN, BPEL, XPDL and EPCs.

**Issues** One consideration that arises with the implementation of the OR-join is providing a form that is able to be used in loops and more complex process models which are not structured in form. The *Structured Synchronizing Merge* cannot be used in these contexts.

**Solutions** Both the *Acyclic Synchronizing Merge* (WCP-37) and the *General Syn-*

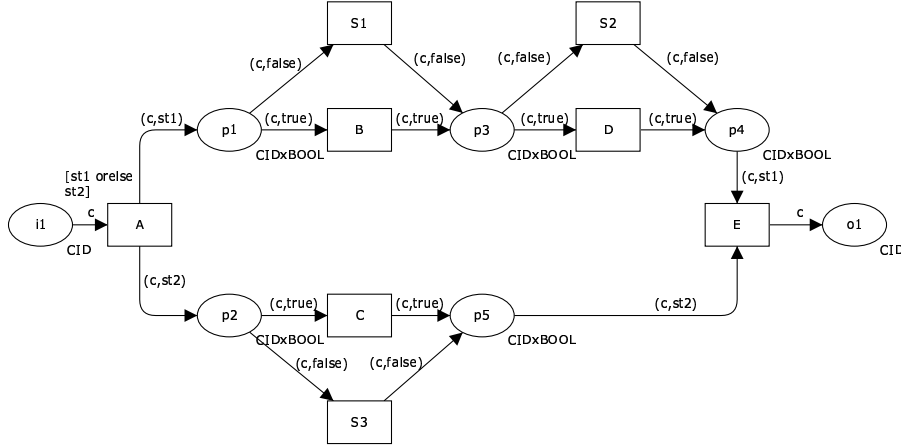


Figure 8: Acyclic synchronizing merge pattern

*chronizing Merge* (WCP-38) are able to be used in unstructured process models. The latter is also able to be used in loops. The *Acyclic Synchronizing Merge* tends to be more attractive from an implementation perspective as it is less computationally expensive than the *General Synchronizing Merge*.

**Evaluation Criteria** Full support for this pattern in an offering is evidenced by the availability of a construct which demonstrates all of the context requirements for this pattern. Any offering which allows the threads of control in any subset of the input branches to the merge to be cancelled before it is triggered achieves a rating of partial support.

### Pattern WCP-8 (Multi-Merge)

**Description** The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

**Synonyms** None.

#### Example

- The *lay\_foundations*, *order\_materials* and *book\_labourer* activities occur in parallel as separate process branches. After each of them completes the *quality\_review* activity is run before that branch of the process completes.

**Motivation** The *Multi-Merge* pattern provides a means of merging distinct branches in a process into a single branch. Although several execution paths are merged, there is no synchronization of control-flow and each thread of control which is currently active in any of the preceding branches will flow unimpeded into the merged branch.

**Context** The operation of this pattern is illustrated in Figure 9. The main condition associated with it is that multiple branches preceding the *Multi-Merge* should result in an active thread of control in the branch after the Multi-Merge. In CPN terms, each incoming token to place **p1** should be preserved. The distinction between this pattern and the *Simple Merge* is that it is possible for more than one incoming branch to be active simultaneously and there is no necessity for place **p1** to be safe.

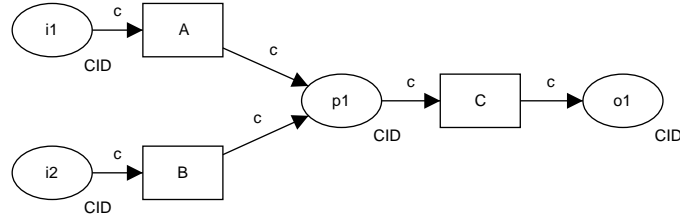


Figure 9: Multi-merge pattern

**Implementation** iPlanet allows the *Multi-Merge* pattern to be implemented by specifying a trigger condition for an activity that allows it to be triggered when any of its incoming routers are triggered. BPMN and XPDL directly implement it via the XOR-join construct and UML 2.0 ADs have an analogue in the form of the Merge-Node construct. EPCs also provide the XOR-join construct however it only expects one incoming thread of control and ignores subsequent simultaneous triggers, hence it does not support the pattern. FLOWer is able to support multiple concurrent threads through dynamic subplans however its highly structured nature does not enable it to provide general support for the *Multi-Merge* pattern. Although COSA is based on a Petri Net foundation, it only supports safe models and hence is unable to fully support the pattern. For example, both A and B in Figure 9 will block if there is a token in place p1. Staffware attempts to maintain a safe process model by coalescing subsequent triggerings of a step whilst it is active into the same thread of control hence it is also unable to support this pattern. This behaviour is quite problematic as it creates a race condition in which all of the execution sequences ABC, BAC, ACBC and BCAB are possible.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it satisfies the context criterion for the pattern. Partial support is awarded to offerings that do not provide support for multiple branches to merge simultaneously or do not provide for preservation of all threads of control where this does occur.

### Pattern WCP-9 (Structured Discriminator)

**Description** The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model. The thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The discriminator construct resets when all incoming branches have been enabled.

**Synonym** 1-out-of-M join.

#### Example

- When handling a cardiac arrest, the *check\_breathing* and *check\_pulse* activities run in parallel. Once the first of these has completed, the *triage* activity is commenced. Completion of the other activity is ignored and does not result in a second instance of the *triage* activity.

**Motivation** The *Discriminator* pattern provides a means of merging two or more distinct branches in a process into a single subsequent branch such that the first of them to complete results in the subsequent branch being triggered, but completions of other incoming branches thereafter have no effect on (and do not trigger) the subsequent branch. As such, the *Discriminator* provides a mechanism for progressing the execution of a process once the first of a series of concurrent activities has completed.

**Context** The *Discriminator* pattern provides a means of merging two or more branches in a workflow and progressing execution of the workflow as rapidly as possible by enabling the subsequent (merged) branch as soon as a thread of control is received on one of the incoming branches. There are five context conditions associated with the use of this pattern:

1. The *Discriminator* is associated with precisely one *Parallel Split* earlier in the process and each of the outputs from the *Parallel Split* is an input to the *Discriminator*;
2. The branches from the *Parallel Split* to the *Discriminator* are structured in form and any splits and merge in the branches are balanced;
3. Each of the incoming branches to the *Discriminator* must only be triggered once prior to it being reset;
4. The *Discriminator* resets (and can be re-enabled) once all of its incoming branches have been enabled precisely once; and
5. Once the *Parallel Split* has been enabled none of the activities in the branches leading to the *Discriminator* can be cancelled before the join has been triggered. The only exception to this is that it is possible for *all* of the activities leading up to the *Discriminator* to be cancelled.

The operation of the *Structured Discriminator* pattern is illustrated in Figure 10. The () notation indicates a simple untyped token. Initially there is such a token in place **p2** (which indicates that the *Discriminator* is ready to be enabled). The first token received at any of the incoming places **i1** to **im** results in the *Discriminator* being enabled and an output token being produced in output place **o1**. An untyped token is also produced in place **p3** indicating that the *Discriminator* has fired but not yet reset. Subsequent tokens received at each of the other input places have no effect on the *Discriminator* (and do not result in any output tokens in place **o1**). Once one token has been received by each input place, the *Discriminator* resets and can be re-enabled once again. This occurs when **m-1** tokens have accumulated at place **p1** allowing the **reset** transition to be enabled.<sup>4</sup>

There are two possible variants of this pattern that arise from relaxing some of the context conditions associated with the *Structured Discriminator* pattern. Both of these improve the applicability of the *Structured Discriminator* pattern whilst retaining its overall behaviour.

First, the *Blocking Discriminator* (WCP-28) removes the requirement that each incoming branch can only be enabled once between *Discriminator* resets. It allows

---

<sup>4</sup>As a general comment, the notation **x'c** on an input arc to a CPN transition means that **x** instances of token **c** are required for the input arc to be enabled.

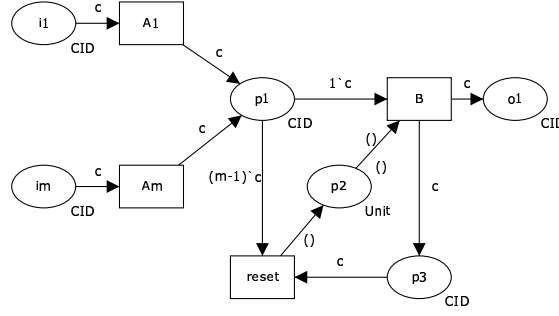


Figure 10: Structured discriminator pattern

each incoming branch to be triggered multiple times although the construct only resets when one triggering has been received on each input branch. It is illustrated in Figure 11 and discussed in further detail on page 54.

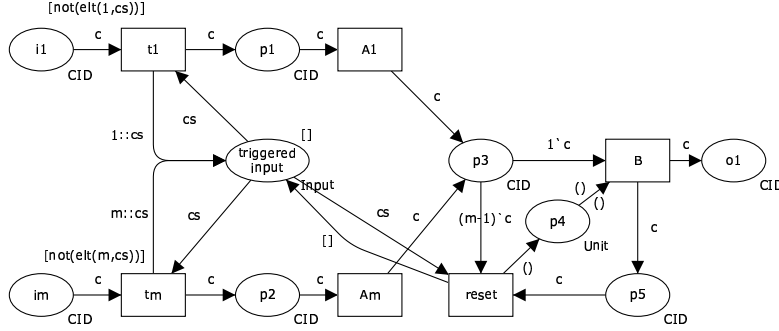


Figure 11: Blocking discriminator pattern

The second alternative, the *Cancelling Discriminator* (WCP-29), improves the efficiency of the pattern further by preventing any subsequent activities in the remaining incoming branches to the *Discriminator* from being enabled once the first branch has completed. Instead the remaining branches are effectively put into a “bypass mode” where any remaining activities are “skipped” hence expediting the reset of the construct. It is illustrated in Figure 12 and discussed in further detail on page 54.

**Implementation** The *Structured Discriminator* can be directly implemented in iPlanet by specifying a custom trigger condition for an activity with multiple incoming routers which only fires when the first router is enabled. BPMN and XPD L potentially support the pattern with a COMPLEX-Join construct however it is unclear how the IncomingCondition for the join is specified. UML 2.0 ADs shares a similar problem with its JoinNode construct. SAP Workflow provides partial support for this pattern via the fork construct although any unfinished branches are cancelled once the first completes.

**Issues** One issue that can arise with the *Structured Discriminator* is that failure to receive input on each of the incoming branches may result in the process instance (and possibly other process instances) deadlocking.

**Solutions** The alternate versions of this pattern provide potential solutions to the is-

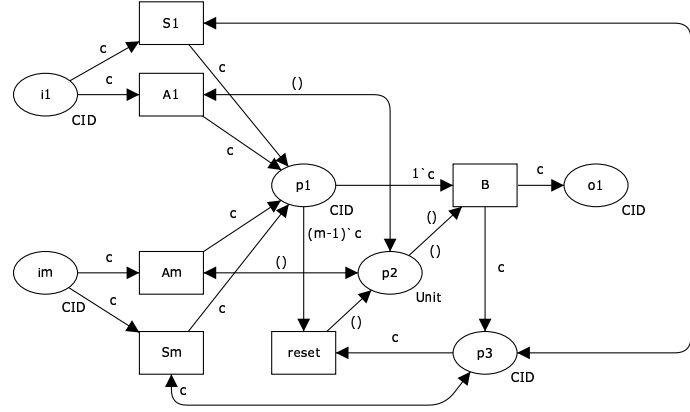


Figure 12: Cancelling discriminator pattern

sue. The *Blocking Discriminator* allows multiple execution threads in a given process instance to be handled by a single *Discriminator* (although a subsequent thread can only trigger the construct when inputs have been received on all incoming branches and the *Discriminator* has reset). The *Cancelling Discriminator* only requires the first thread of control to be received in an incoming branch. Once this has been received, the remaining branches are effectively put into “bypass” mode and any remaining activities in those branches that have not already been commenced are skipped allowing the discriminator to be reset as soon as possible.

**Evaluation Criteria** An offering achieves full support if it satisfies the context criteria for the pattern. It rates as partial support if the *Discriminator* can reset without all activities in incoming branches having run to completion.

### 3.3.3 Structural Patterns

Structural patterns characterise design restrictions that specific workflow languages may have on the form of process model that they are able to represent and how these models behave at runtime. There are two main areas that are of interest in structural terms: (1) the form of cycles or loops that can be represented within the process model and (2) whether the termination of a process instance must be explicitly captured within the process model.

Looping is a common construct that arises during process modelling in situations where individual activities or groups of activities must be repeated. Three distinct forms of repetition can be identified: *Arbitrary Cycles*, *Structured Loops* and *Recursion*. In classical programming terms, these correspond to the notions of (1) loops based on `goto` statements, which tend to be somewhat unstructured in format with repetition achieved by simply moving the thread of execution to a different part of the process model, possibly repeatedly, (2) more structured forms of repetition based on dedicated programmatic constructs such as `while...do` and `repeat...until` statements<sup>5</sup> and (3) repetition based on self-invocation. All of these structural forms of repetition have distinct characterizations and they form the basis for the *Arbitrary*

<sup>5</sup>The comparison of arbitrary loops to `goto` statements is a bit misleading. Note that in most graphical modelling languages, it is possible to connect one node to another (independent of loops). This should not be considered as “sloppy modelling”, but rather as a feature!

*Cycles* (WCP-10), *Structured Loop* (WCP-21) and *Recursion* (WCP-22) patterns. In the original set of patterns there was no consideration of structured loops or recursive iteration.

Another structural consideration associated with individual process modelling formalisms is whether a process instance should simply end when there is no remaining work to be done or whether a specific construct should exist in the process model to denote the termination of a process instance. The first of these alternatives is arguably a closer analogue to the way in which many business processes actually operate. It is described in the form of the *Implicit Termination* pattern (WCP-11). A second pattern *Explicit Termination* pattern (WCP-43) has been introduced to recognize the fact that many workflow languages opt for a concrete form of denoting process endpoints.

### Pattern WCP-10 (Arbitrary Cycles)

**Description** The ability to represent cycles in a process model that have more than one entry or exit point.

**Synonyms** Unstructured loop, iteration, cycle.

**Example** Figure 13 provides an illustration of the pattern with two entry points: p3 and p4.

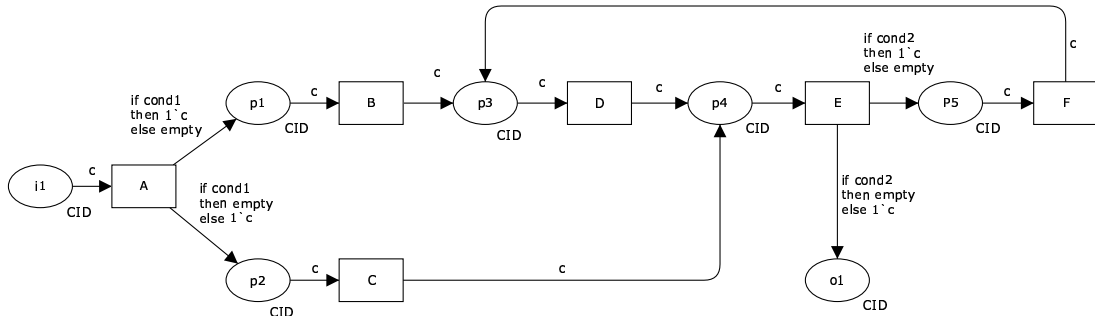


Figure 13: Arbitrary cycles pattern

**Motivation** The *Arbitrary Cycles* pattern provides a means of supporting repetition in a process model in an unstructured way without the need for specific looping operators or restrictions on the overall format of the process model.

**Context** There are no specific context conditions associated with the inclusion of arbitrary cycles in a process model other than the obvious requirement that the process model is able to support cycles (i.e. it is not block structured).

**Implementation** Staffware, COSA, iPlanet, FileNet, BPMN, XPDL, UML 2.0 ADs and EPCs are all capable of capturing the arbitrary cycles pattern. Block structured offerings such as WebSphere MQ, FLOWer, SAP Workflow and BPEL are not able to represent arbitrary process structures.

**Issues** The unstructured occurrences of the *Arbitrary Cycles* pattern are difficult to capture in many types of workflow products, particularly those that implement structured process models.

**Solutions** In some situations it is possible to transform process models containing *Arbitrary Cycles* into structured workflows, thus allowing them to be captured in



structured workflow products. Further details on the types of process models that can be transformed and the approaches to doing so can be found in [KtHB00] and [Kie03].

**Evaluation Criteria** An offering achieves full support for the pattern if it is able to capture unstructured cycles that have more than one entry or exit point.

### Pattern WCP-11 (Implicit Termination)

**Description** A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future.

**Synonyms** None.

**Example** N/A.

**Motivation** The rationale for this pattern is that it represents the most realistic approach to determining when a process instance can be designated as complete. This is when there is no remaining work to be completed as part of it and it is not possible that work items will arise at some future time.

**Context** There are no specific context considerations associated with this pattern.

**Implementation** Staffware, WebSphere MQ, FLOWer, FileNet, BPEL, BPMN, XPDL, UML 2.0 ADs and EPCs support this pattern. iPlanet requires processes to have a unique end node. COSA terminates a process instance when a specific type of end node is reached.

**Issues** Where an offering does not directly support this pattern, the question arises as to whether it can implement a process model which has been developed based on the notion of implicit termination.

**Solutions** For simple process models, it may be possible to indirectly achieve the same effect by replacing all of the end nodes for a process with links to an OR-join which then links to a single final node. However, it is less clear for more complex process models involving multiple instance activities whether they are always able to be converted to a model with a single terminating node. Potential solutions to this are discussed at length in [KtHvdA03].

It is worthwhile noting that some languages do not offer this construct on purpose: the *Implicit Termination* pattern makes it difficult (or even impossible) to distinguish proper termination from deadlock! Additionally, workflows without explicit endpoints are more difficult to use in compositions.

**Evaluation Criteria** An offering achieves full support for this pattern if process (or sub-process) instances terminate when there are no remaining activities to be completed now or at any time in the future and the process instance is not in deadlock.

### 3.3.4 Multiple Instance Patterns

Multiple instance patterns describe situations where there are multiple threads of execution active in a process model which relate to the same activity (and hence share the same implementation definition). Multiple instances can arise in three situations:

1. An activity is able to initiate multiple instances of itself when triggered (we denote this form of activity a *multiple instance* activity);
2. A given activity is initiated multiple times as a consequence of it receiving several independent triggerings, e.g. as part of a loop or in a process instance in which there are several concurrent threads of execution as might result from a *Multi-Merge* for example; and
3. Two or more activities in a process share the same implementation definition. This may be the same activity definition in the case of a multiple instance activity or a common sub-process definition in the case of a block activity. Two (or more) of these activities are triggered such that their executions overlap (either partially or wholly).

Although all of these situations potentially involve multiple concurrent instances of an activity or sub-process, it is the first of them in which we are most interested as they require the triggering and synchronization of multiple concurrent activity instances. This group of patterns focusses on the various ways in which these events can occur.

Similar to the differentiation introduced in the Advanced Branching and Synchronization Patterns to capture the distinction between the *Discriminator* and the *Partial Join* pattern variants, three new patterns have been introduced to recognize alternative operational semantics for multiple instances. These are the *Static Partial Join for Multiple Instances* (WCP-34), the *Cancelling Partial Join for Multiple Instances* (WCP-35) and the *Dynamic Partial Join for Multiple Instances* (WCP-36), each of which is discussed in detail in Section 4.

### Pattern WCP-12 (Multiple Instances without Synchronization)

**Description** Within a given process instance, multiple instances of an activity can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion.

**Synonyms** Multi threading without synchronization, spawn off facility.

#### Example

- A list of traffic infringements is received by the Transport Department. For each infringement on the list an *Issue-Infringement-Notice* activity is created. These activities run to completion in parallel and do not trigger any subsequent activities. They do not need to be synchronized at completion.

**Motivation** This pattern provides a means of creating multiple instances of a given activity. It is particularly suited to situations where the number of individual activities required is known before the spawning action commences, the activities can execute independently of each other and no subsequent synchronization is required.

**Context** There are two possible variants in the way in which this pattern can operate. The first is illustrated by Figure 14 in which the **create instance** activity runs within a loop and the new activity instances are created sequentially. Place **p2** indicates the number of instances required and is decremented as each new instance is created. New instances can only be created when the token in **p2** > 0 – the guard on the **create instance** activity ensures this is the case. When all instances have

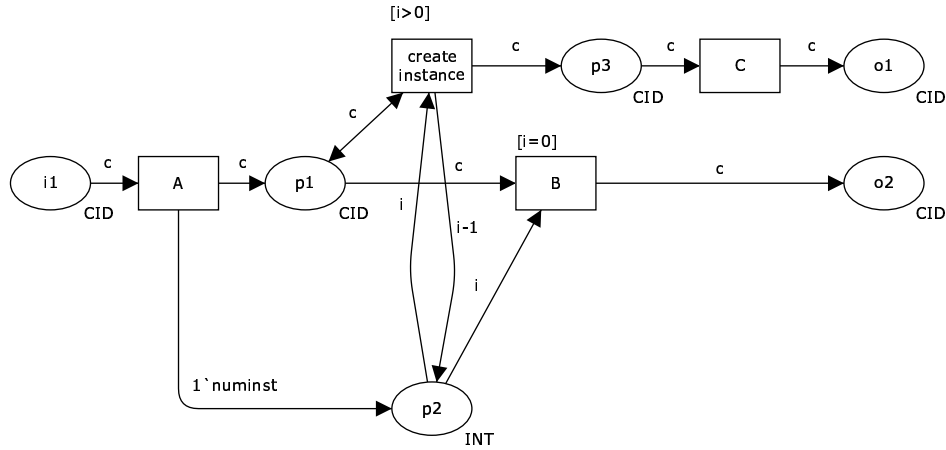


Figure 14: Multiple instances without synchronization pattern - 1

been created, the next activity (B) can be enabled – again the guard on activity B ensures this is also the case.

In Figure 15, the activity instances are all created simultaneously. In both variants, it is a requirement that the number of new instances required is known before the creation activity commences. It is also assumed that activity instances can be created that run independently (and in addition to the thread of control which started them) and that they do not require synchronizing as part of this construct.

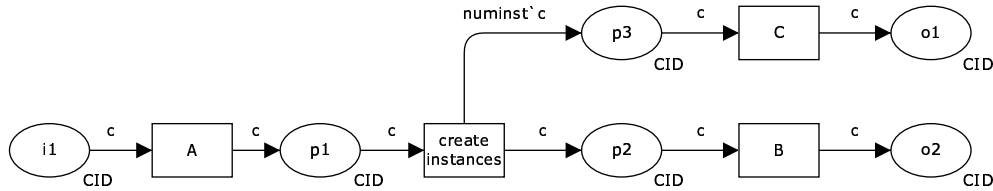


Figure 15: Multiple instances without synchronization pattern - 2

There are two context conditions associated with this pattern: (1) each of the multiple instance activities that are created must execute within the context of the process instance from which they were started (i.e. they must share the same case id and have access to the same data elements) and (2) each of the multiple instance activities must execute independently from and without reference to the activity that started them.

**Implementation** Most offerings – COSA, iPlanet, BPEL, BPMN, XPD and UML 2.0 ADs – support the sequential variant of this pattern (as illustrated in Figure 14) with the activity creation occurring within a loop. SAP Workflow also do so, but with the limitation that a new process instance is started for each activity instance invoked. BPMN also supports the second variant, as do Staffware and FLOWer, and they provide the ability to create the required number of activity instances simultaneously.

**Issues** Where an offering provides support for this pattern, one issue that can potentially arise is how the various threads of execution might be synchronized at some future point in the process.

**Solutions** This is potentially problematic as it is likely that the individual threads of execution may ultimately flow down the same path and most offerings do not provide a construct for this form of synchronization. In recognition of this need (and the fact that some languages do provide such a facility), the *Thread Merge* (WCP-41) and *Thread Split* (WCP-42) patterns have been introduced.

**Evaluation Criteria** An offering achieves full support if it satisfies the context requirements for the pattern. Where the newly created activity instances run in a distinct process instance to the activity that started them or they cannot access the same data elements as the parent activity, the offering achieves only partial support.

### Pattern WCP-13 (Multiple Instances with *a priori* Design-Time Knowledge)

**Description** Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the activity instances at completion before any subsequent activities can be triggered.

**Synonyms** None.

#### Example

- The Annual Report must be signed by all six of the Directors before it can be issued.

**Motivation** This pattern provides the basis for concurrent execution of a nominated activity a predefined number of times. It also ensures that all activity instances are complete before subsequent activities are initiated.

**Context** Similar to WCP-12, the *Multiple Instances without Synchronization* pattern, there are both sequential and simultaneous variants of this pattern illustrated in Figures 16 and 17 respectively. In both figures, activity C is the one that executes multiple times.

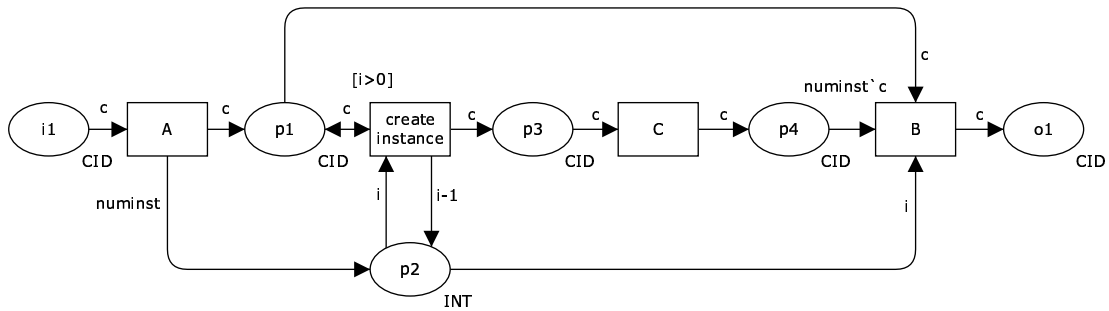


Figure 16: Multiple instances with *a priori* design-time knowledge - 1

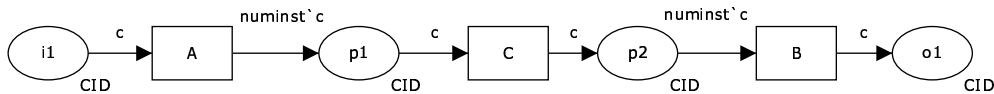


Figure 17: Multiple instances with *a priori* design-time knowledge - 2

There are three context conditions associated with this pattern: (1) the number of activity instances required must be specified in the design-time process model, (2) it must be possible for the activity instances to execute concurrently (although it is not necessarily required that they do all execute in parallel) and (3) all activity instances must complete before subsequent activities in the process can be triggered.

**Implementation** In order to implement this pattern, an offering must provide a specific construct in the process model that is able to denote the specific number of concurrent activity instances that are required. Staffware, FLOWer, SAP Workflow and UML 2.0 ADs support the simultaneous variant of the pattern through the use of dynamic subprocedure, dynamic subplan, multi-line container element and ExpansionRegion constructs respectively. BPMN and XPDL support both options via the multi-instance loop activity construct with the `MLOrdering` attribute supporting both sequential and parallel values depending on whether the activities should be started one-by-one or all together. Unlike other BPEL offerings which do not support this pattern, Oracle BPEL provides a `<flowN>` construct that enables the creation of multiple concurrent instances of an activity.

**Issues** Many offerings provide a work-around for this pattern by embedding some form of activity invocation within a loop. These implementation approaches have two significant problems associated with them: (1) the activity invocations occur at discrete time intervals and it is possible for the individual activity instances to have potentially distinct states at the time they are invoked (i.e. the activities do not need to be executed in sequence and can be handled concurrently) and (2) there is no consideration of the means by which the distinct activity instances will be synchronized. These issues, together with the necessity for the designer to effectively craft the pattern themselves (rather than having it provided by the offering) rule out this form of implementation from being considered as satisfying the requirements for full support.

**Solutions** One possibility that exists where this functionality is not provided by an offering but an analogous form of operation is required is to simply replicate the activity in the process-model. Alternatively a solution based on iteration can be utilized.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern. Although work-arounds are possible which achieve the same behaviour through the use of various constructs within an offering such as activity replication or loops, they have a number of shortcomings and are not considered to constitute support for the pattern.

#### **Pattern WCP-14 (Multiple Instances with *a priori* Run-Time Knowledge)**

**Description** Within a given process instance, multiple instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the activity instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent activities can be triggered.

**Synonyms** None.

## Examples

- When diagnosing an engine fault, the *check-sensor* activity can run multiple times depending on the the number of error messages received. Only when all messages have been processed, can the *identify-fault* activity be initiated;
- In the review process for a journal paper submitted to a journal, the *review paper* activity is executed several times depending on the content of the paper, the availability of referees and the credentials of the authors. The review process can only continue when all reviews have been returned;
- When dispensing a prescription, the *weigh compound* activity must be completed for each ingredient before the preparation can be compounded and dispensed.

**Motivation** The *Multiple Instances with a priori Run-Time Knowledge* pattern provides a means of executing multiple instances of a given activity in a synchronized manner with the determination of exactly how many instances will be created being deferred to the latest possible time before the first of the activities is started.

**Context** As with other multiple instance patterns, there are two variants of this pattern depending on whether the instances are created sequentially or simultaneously as illustrated in Figures 18 and 19. In both cases, the number of instances of activity C to be executed (indicated in these diagrams by the variable *numinst*) is communicated at the same time that the thread of control is passed for the process instance.

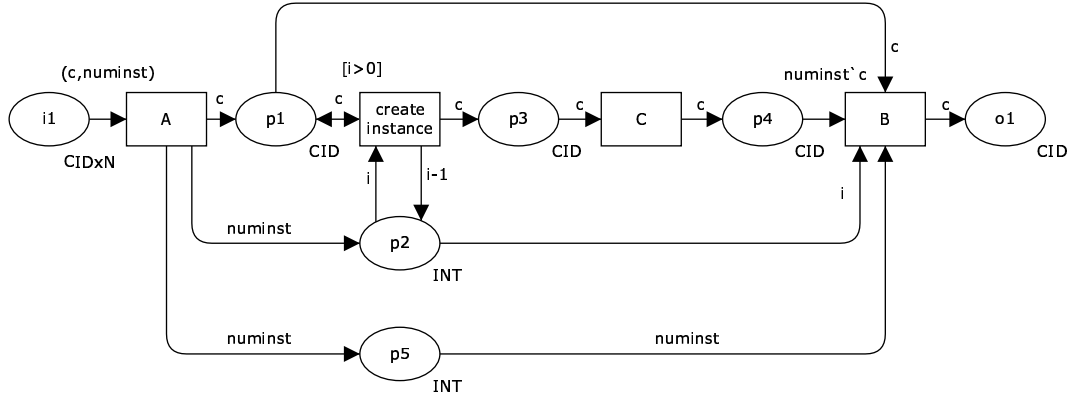


Figure 18: Multiple instances with a priori run-time knowledge pattern - 1

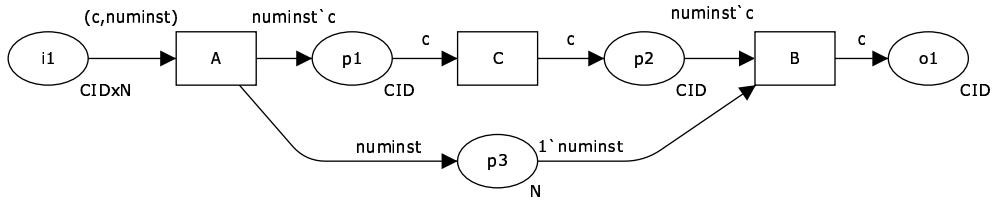


Figure 19: Multiple instances with a priori run-time knowledge pattern - 2

There are three context conditions associated with this pattern: (1) the number of activity instances required must be known at run-time prior to the invocation of the multiple instance activity, (2) it must be possible for the activity instances to

execute concurrently (although it is not necessarily required that they do all execute in parallel) and (3) all activity instances must complete before subsequent activities in the process can be triggered.

**Implementation** Staffware, FLOWer and UML 2.0 ADs support the simultaneous variant of the pattern through the use of dynamic subplan and ExpansionRegion constructs respectively. BPMN and XPDL support both options via the multi-instance loop activity construct. In the case of FLOWer, BPMN and XPDL, the actual number of instances required is indicated through a variable passed to the construct at runtime. For UML 2.0 ADs, the ExpansionRegion construct supports multiple instantiations of an activity based on the number of instances of a defined data element(s) passed at run-time. Oracle BPEL supports the pattern via its (unique) <flowN> construct.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern.

### **Pattern WCP-15 (Multiple instances without *a priori* run-time knowledge)**

**Description** Within a given process instance, multiple instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent activities can be triggered.

**Synonyms** None.

#### **Example**

- The despatch of an *oil rig* from factory to site involves numerous *transport shipment* activities. These occur concurrently and although sufficient activities are started to cover initial estimates of the required transport volumes, it is always possible for additional activities to be initiated if there is a shortfall in transportation requirements. Once the whole oil rig has been transported, and all *transport shipment* activities are complete, the next activity (*assemble rig*) can commence.

**Motivation** This pattern is an extension to WCP-14 *Multiple Instances with a priori Run-Time Knowledge* which defers the need to determine how many concurrent instances of the activity are required until the last possible moment – either when the final join construct fires or the last of the executing instances completes. It offers more flexibility in that additional instances can be created “on-the-fly” without any necessary change to the process model or the synchronization conditions for the activity.

**Context** Similar to other multiple instance patterns, there are two variants to this pattern depending on whether the initial round of instances are started sequentially

or simultaneously. These scenarios are depicted in Figures 20 and 21. It should be noted that it is possible to add additional instances of activity C in both of these implementations via the **add instance** transition at any time up until all instances have completed and the join associated with them has fired triggering the subsequent activity (B).

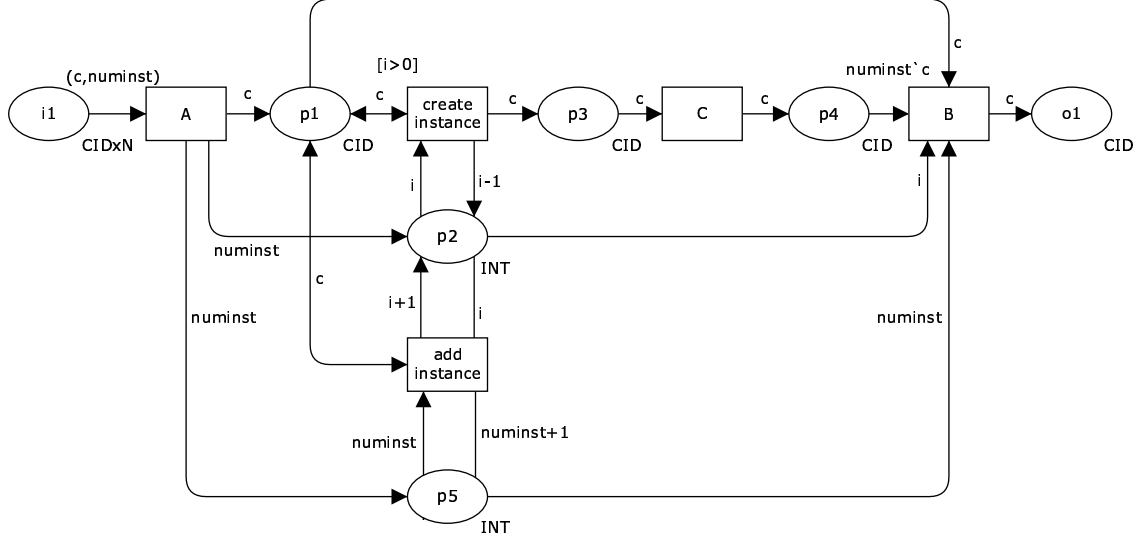


Figure 20: Multiple instances without a priori run-time knowledge pattern - 1

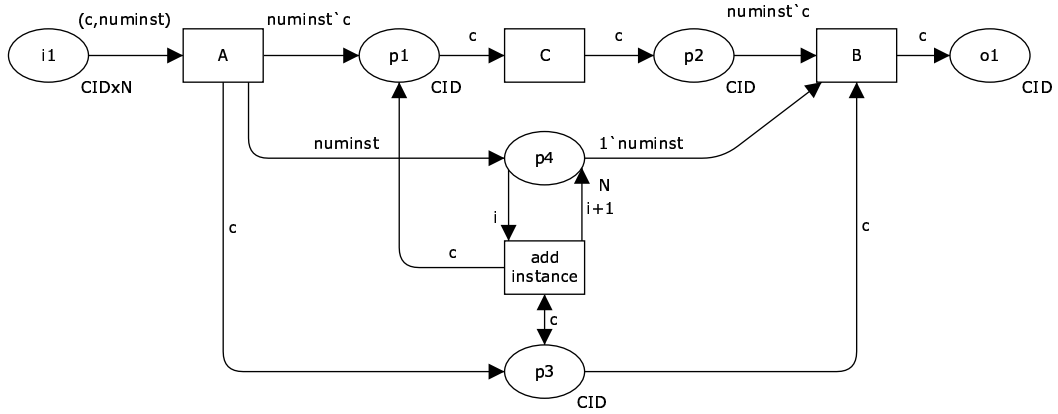


Figure 21: Multiple instances without a priori run-time knowledge pattern - 2

There are three context conditions associated with this pattern: (1) the number of activity instances required must be known at run-time prior to the completion of the multiple instance activity (note that the final number of instances does not need to be known when initializing the MI activity), (2) it must be possible for the activity instances to execute concurrently (although it is not necessarily required that they do all execute in parallel) and (3) all activity instances must complete before subsequent activities in the process can be triggered.

**Implementation** Only one of the offerings examined – FLOWer – provides direct support for this pattern. It does this through the dynamic subplan construct.



**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern.

### 3.3.5 State-based Patterns

State-based patterns reflect situations for which solutions are most easily accomplished in process languages that support the notion of state. In this context, we consider the state of a process instance to include the broad collection of data associated with current execution including the status of various activities as well as process-relevant working data such as activity and case data elements.

The original patterns include three patterns in which the current state is the main determinant in the course of action that will be taken from a control-flow perspective. These are: *Deferred Choice* (WCP-16), where the decision about which branch to take is based on interaction with the operating environment, *Interleaved Parallel Routing* (WCP-17), where two or more sequences of activities are undertaken on an interleaved basis such that only one activity instance is executing at any given time and *Milestone* (WCP-18), where the enabling of a given activity only occurs where the process is in a specific state.

In recognition of further state-based modelling scenarios, four new patterns have also been identified and are discussed in detail in Section 4. These are: *Critical Section* (WCP-39), which provides the ability to prevent concurrent execution of specific parts of a process, *Interleaved Routing* (WCP-40), which denotes situations where a group of activities can be executed sequentially in *any* order, and *Thread Merge* (WCP-41) and *Thread Split* (WCP-42) which provide for coalescence and divergence of distinct threads of control along a single branch.

#### Pattern WCP-16 (Deferred Choice)

**Description** A point in a workflow process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches present possible future courses of execution. The decision is made by initiating the first activity in one of the branches i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn.

**Synonyms** External choice, implicit choice, deferred XOR-split.

#### Examples

- At the commencement of the *Resolve complaint* process, there is a deferred choice between the *Initial customer contact* activity and the *Escalate to manager* activity. The *Initial customer contact* is initiated when it is started by a customer services team member. The *Escalate to manager* activity commences 48 hours after the process instance commences. Once one of these activities is initiated, the other is withdrawn.
- Once a customer requests an *airbag shipment*, it is either picked up by the *postman* or a *courier driver* depending on which is available to visit the customer site first.

**Motivation** The *Deferred Choice* pattern provides the ability to defer the *moment of choice* in a process, i.e. the moment as to which one of several possible courses of

action should be chosen is delayed to the last possible time and is based on factors external to the process instance (e.g. incoming messages, environment data, resource availability, timeouts etc.). Up until the point at which the decision is made, any of the alternatives presented represent viable courses of future action.

**Context** The operation of this pattern is illustrated in Figure 22. The *moment of choice* is signified by place p1. Either activity B or C represent valid courses of action but only one of them can be chosen.

It is a context condition of this pattern that once one of the possible alternative courses of action is chosen, any possible actions associated with other branches are immediately withdrawn.

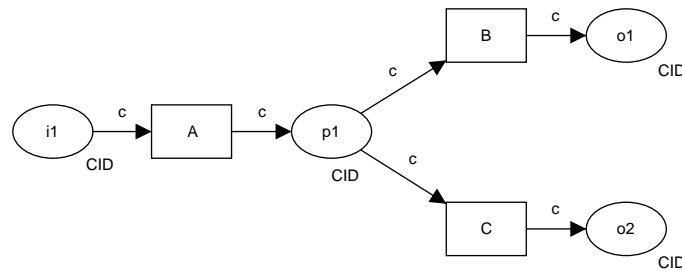


Figure 22: Deferred choice construct

**Implementation** This is a complex pattern and it is interesting to see that only those offerings that can claim some sort of token-based underpinning are able to successfully support it. COSA is based on a Petri-Net foundation and can implement the pattern in much the same way as it is presented in Figure 22. BPEL provides support for it via the <pick> construct, BPMN through the event-based gateway construct, XPDL using the XOREVENT-split construct and UML 2.0 ADs using a ForkNode followed by a set of AcceptSignal actions, one preceding each action in the choice. In the case of the latter three offerings, the actual choice is made based on message-based event interactions. FLOWer does not directly provide a notion of state but it provides several ways of supporting this pattern through the use of user and system decisions on plan types and also by using arc guards that evaluate to NIL in conjunction with data elements to make the decision as to which branch is selected. FileNet provides partial support for the pattern as it only allows for withdrawal of timer-based branches not of all branches other than the one selected for execution.

**Issue** None identified.

**Solution** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern. If there are any restrictions on which branches can be selected or withdrawn, then the offering is rated as having partial support.

### Pattern WCP-17 (Interleaved Parallel Routing)

**Description** A set of activities has a partial ordering defining the requirements with respect to the order in which they must be executed. Each activity in the set must be executed once and they can be completed in any order that accords with the partial

order. However, as an additional requirement, no two activities can be executed at the same time (i.e. no two activities can be active for the same process instance at the same time).

**Synonyms** None.

### Example

- When despatching an order, the *pick goods*, *pack goods* and *prepare invoice* activities must be completed. The *pick goods* activity must be done before the *pack goods* activity. The *prepare invoice* activity can occur at any time. Only one of these activities can be done at any time for a given order.

**Motivation** The *Interleaved Parallel Routing* pattern offers the possibility of relaxing the strict ordering that a process usually imposes over a set of activities. Note that *Interleaved Parallel Routing* is related to mutual exclusion, i.e. a semaphore makes sure that activities are not executed at the same time without enforcing a particular order.

**Context** Figure 23 provides an example of interleaved parallel routing. Place p3 enforces that activities B, C and D be executed in some order. In this example, the permissible activity orderings are: ABDCE, ABCDE and ACBDE.

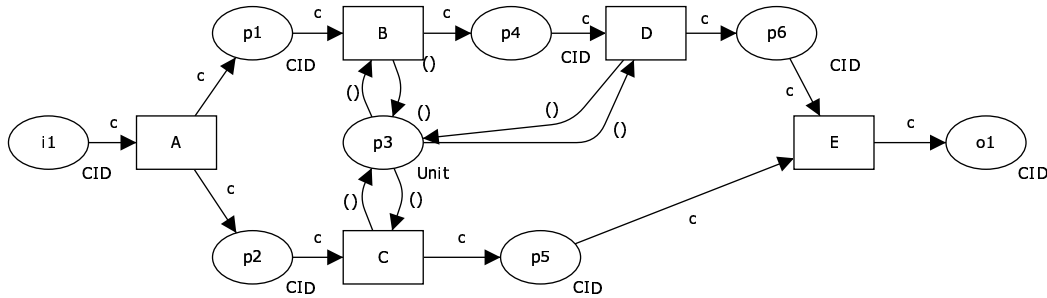


Figure 23: Interleaved parallel routing pattern

There are three context conditions associated with this pattern: (1) for a given process instance, no two activities from the set of activities subject to interleaved parallel routing may be executed at the same time, (2) there must be some (partial) ordering defined between the activities and (3) activities must be initiated and completed on a sequential basis, it is not possible to suspend one activity during its execution to work on another.

If the second condition is not satisfied, then the scenario is actually one of *Interleaved Routing* and is described by pattern WCP-40 on page 73.

**Implementation** In order to effectively implement this pattern, an offering must have an integrated notion of state that is available during execution of the control-flow perspective. COSA has this from its Petri-Net foundation and is able to directly support the pattern. Other offerings lack this capability and hence are not able to directly support this pattern. BPEL (although surprisingly not Oracle BPEL) can indirectly achieve similar effects using serializable scopes within the context of a <pick> construct although only activities in the same block can be included within it. It also has the shortcoming that every permissible execution sequence of interleaved activities must be explicitly modelled. FLOWer has a distinct foundation to that inherent in other workflow products in which all activities in a case are always allocated to the

same resource for completion hence interleaving of activity execution is guaranteed, however it is also possible for a resource to suspend an activity during execution to work on another hence the context conditions for this pattern are not fully satisfied.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it is able to satisfy the context criteria for the pattern. It achieves a partial support rating if there are any limitations on the set of activities that be interleaved or if activities can be suspended during execution.

### Pattern WCP-18 (Milestone)

**Description** An activity is only enabled when the process instance (of which it is part) is in a specific state (typically in a parallel branch). The state is assumed to be a specific execution point (also known as a *milestone*) in the process model. When this execution point is reached the nominated activity can be enabled. If the process instance has progressed beyond this state, then the activity cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger.

**Synonyms** Test arc, deadline, state condition, withdraw message.

#### Example

- Most budget airlines allow the routing of a booking to be changed providing the ticket has not been issued;
- The *enrol student* activity can only execute whilst new enrolments are being accepted. This is after the *open enrolment* activity has completed and before the *close off* enrolment activity commences.

**Motivation** The *Milestone* pattern provides a mechanism for supporting the conditional execution of an activity or sub-process (possibly on a repeated basis) where the process instance is in a given state. The notion of state is generally taken to mean that control-flow has reached a nominated point in the execution of the process instance (i.e. a *milestone*). As such, it provides a means of synchronizing two distinct branches of a process instance, such that one branch cannot proceed unless the other branch has reached a specified state.

**Context** The nominal form of the milestone pattern is illustrated by Figure 24. Activity A cannot be enabled when it receives the thread of control unless the other branch is in state **p1** (i.e. there is a token in place **p1**). This situation presumes that the process instance is either in state **p1** or will be at some future time. It is important to note that the repeated execution of A does not influence the top parallel branch.

Note that A can only occur if there is a token in **p1**. Hence a milestone may have a potential deadlock. There are at least two ways of avoiding this. First of all, it is possible to define an alternative activity for A which takes a token from the input place(s) of A without taking a token from **p1**. One can think of this activity as a time-out or a skip activity. This way the process does not get stuck if C occurs before A. Moreover, it is possible to delay the execution of C until the lower branch finishes.

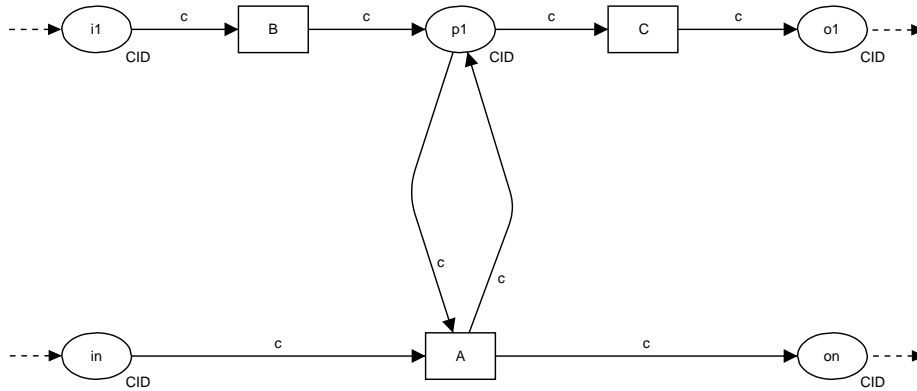


Figure 24: Milestone pattern

Note that in both cases **A** may be optional (i.e. not execute at all) or can occur multiple times because the token in **p1** is only tested and not removed.

**Implementation** The necessity for an inherent notion of state within the process model means that the *Milestone* pattern is not widely supported. Of the offerings examined, only COSA is able to directly represent it. FLOWer offers indirect support for the pattern through the introduction of a data element for each situation in which a milestone is required. This data element can be updated with a value when the milestone is reached and the branch which must test for the *Milestone* achievement can do so using the FLOWer milestone construct. Note that this is only possible in a data-driven system like FLOWer. It is not possible to use variables this way in a classical control-flow driven system because a “busy wait” would be needed to constantly inspect the value of this variable. (Note that FLOWer only re-evaluates the state after each change with respect to data elements).

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that allows the execution of a given activity to be dependent on the process instance being in some predefined state.

### 3.3.6 Cancellation Patterns

Several of the patterns in previous sections (e.g. (WCP-6) *Structured Synchronizing Merge* and (WCP-9) *Structured Discriminator*) have variants that utilize the concept of activity cancellation where enabled or active activity instances are withdrawn. Various forms of exception handling in processes are also based on cancellation concepts. This section presents two cancellation patterns – *Cancel Activity* (WCP-19) and *Cancel Case* (WCP-20).

Three new cancellation patterns have also been identified *Cancel Region* (WCP-25), *Cancel Multiple Instance Activity* (WCP-26) and *Complete Multiple Instance Activity* (WCP-27). These are discussed in Section 4.

#### Pattern WCP-19 (Cancel Activity)

**Description** An enabled activity is withdrawn prior to it commencing execution. If the activity has started, it is disabled and, where possible, the currently running instance is halted and removed.

**Synonym** Withdraw activity.

**Examples**

- The *assess damage* activity is undertaken by two insurance assessors. Once the first assessor has completed the activity, the second is cancelled;
- The purchaser can cancel their *building inspection* activity at any time before it commences.

**Motivation** The *Cancel Activity* pattern provides the ability to withdraw an activity which has been enabled. This ensures that it will not commence execution.

**Context** The general interpretation of the *Cancel Activity* pattern is illustrated by Figure 25. The trigger which has enabled activity B is removed, preventing the activity from proceeding.

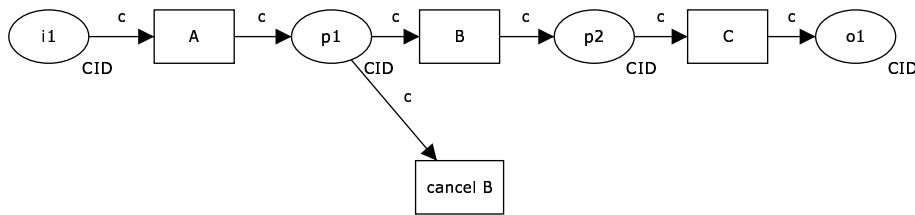


Figure 25: Cancel activity pattern - 1

There is also a second variant of the pattern where the activity has already commenced execution but has not yet completed. This scenario is shown in Figure 26, where an activity which has been enabled or is currently executing can be cancelled. It is important to note for both variants that cancellation is not guaranteed and it is possible that the activity will continue executing to completion. In effect, the cancellation vs continuation decision operates as a deferred choice with a race condition being set up between the cancellation event and the much slower activity of resources responding to work assignment. For all practical purposes, it is much more likely that the cancellation will be effected rather than the activity being continued.

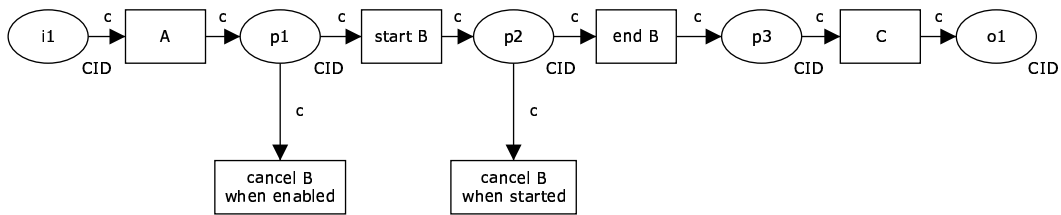


Figure 26: Cancel activity pattern - 2

Where guaranteed cancellation is required, the implementation of activities should take the form illustrated in Figure 27. The decision to cancel activity B can only be made after it has been enabled and prior to it completing. Once this decision is made, it is not possible for the activity to progress any further.

For obvious reasons, it is not possible to cancel an activity which has not been enabled (i.e. there is no “memory” associated with the action of cancelling an activity in the way that there is for triggers) nor is it possible to cancel an activity which has already completed execution.

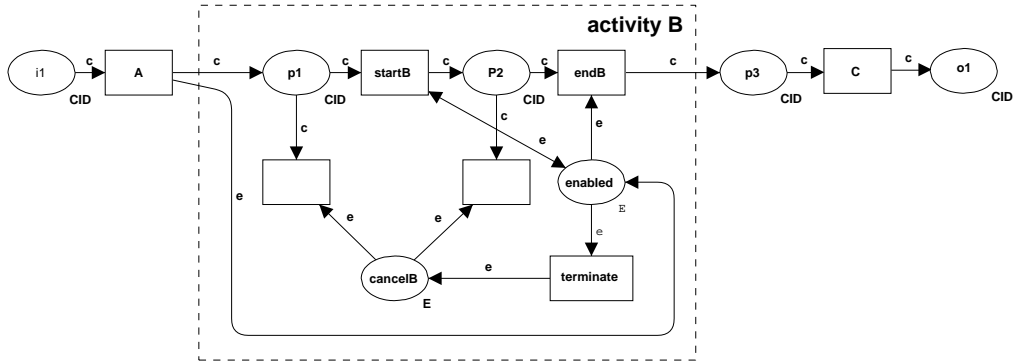


Figure 27: Cancel activity pattern with guaranteed termination

**Implementation** The majority of the offerings examined provide support for this pattern within their process models. Most support the first variant as illustrated in Figure 25: Staffware does so with the withdraw construct, COSA allows tokens to be withdrawn from the places before activities, iPlanet provides the AbortActivity method, FileNet provides the <Terminate Branch> construct and SAP Workflow provides the process control step for this purpose although it has limited usage. BPEL supports the second variant via fault compensation handlers attached to activities, as do BPMN and XPD L using error type triggers attached to the boundary of the activity to be cancelled. UML 2.0 ADs provide a similar capability by placing the activity to be cancelled in an interruptible region triggered by a signal or another activity. FLOWer does not directly support the pattern although activities can be skipped and redone.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support for the pattern if it provides the ability to denote activity cancellation within a process model. If there are any side-effects associated with the cancellation (e.g. forced completion of other activities, the cancelled activity being marked as complete), then the offering is rated as having partial support.

### Pattern WCP-20 (Cancel Case)

**Description** A complete process instance is removed. This includes currently executing activities, those which may execute at some future time and all sub-processes. The process instance is recorded as having completed unsuccessfully.

**Synonym** Withdraw case.

### Examples

- During an *insurance claim* process, it is discovered that the policy has expired and, as a consequence, all activities associated with the particular process instance are cancelled;
- During a *mortgage application*, the purchaser decides not to continue with a house purchase and withdraws the application.

**Motivation** This pattern provides a means of halting a specified process instance and withdrawing any activities associated with it.

**Context** Cancellation of an entire case involves the disabling of all currently enabled activities. Figure 28 illustrates one scheme for achieving this. It is based on the identification of all possible sets of states that the process may exhibit for a process instance. Each combination has a transition associated with it (illustrated by C1, C2.. etc) that disables all enabled activities. Where cancellation of a case is enabled, it is assumed that precisely one of the cancelling transitions (i.e. C1, C2...) will fire cancelling all necessary enabled activities. To achieve this, it is necessary that none of the cancelling transitions represent a state that is a superset of another possible state, otherwise tokens may be left behind after the cancellation.

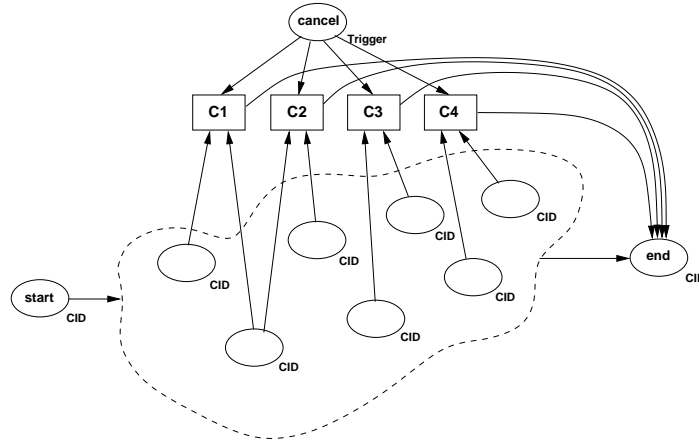


Figure 28: Cancel case pattern - 1

An alternative scheme is presented in Figure 29, where every state has a set of cancellation transitions associated with it (illustrated by C1, C2 .. etc.). When the cancellation is initiated, these transitions are enabled for a very short time interval (in essence the difference between time  $t$  and  $t + \epsilon$  where  $\epsilon$  is a time interval approaching zero), thus effecting an instantaneous cancellation for a given state that avoids the potential deadlocks that might arise with the approach shown in Figure 28.

A more general approach to cancellation is illustrated in Figure 30. This may be used to cancel individual activities, regions or even whole cases. It is premised on the creation of an alternative “bypass” activity for each activity in a process that may need to be cancelled. When a cancellation is initiated, the case continues processing but the “bypass” activities are executed rather than the normal activities, so in effect no further work is actually achieved on the case.

There is an important context condition associated with this pattern: cancellation of a executing case must be viewed as unsuccessful completion of the case. This means that even though the case was terminated in an orderly manner, perhaps even with tokens reaching its final end state, this should not be interpreted in any way as a successful outcome. For example, where a log is kept of events occurring during process execution, the case should be recorded as incomplete or cancelled.

**Implementation** There is reasonable support for this pattern amongst the offerings examined. SAP Workflow provides the process control step for this purpose, FileNet provides the <Terminate Process> construct, BPEL provide the <terminate> construct, BPMN and XPDL provide support by including the entire process in a trans-



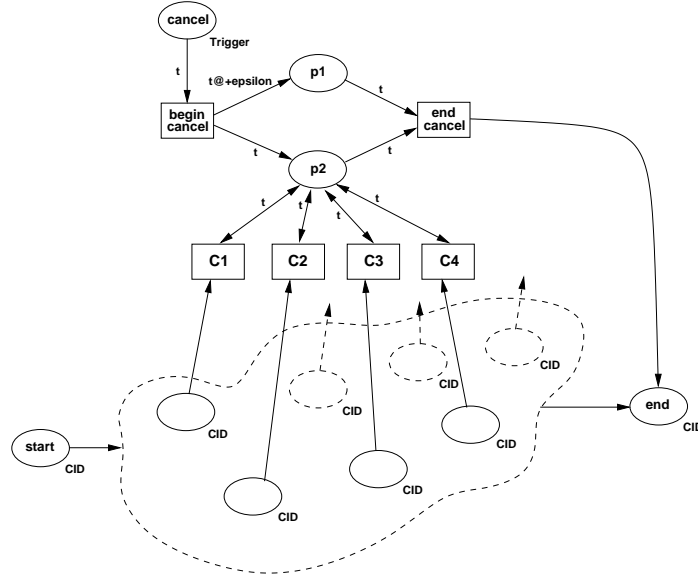


Figure 29: Cancel case pattern - 2

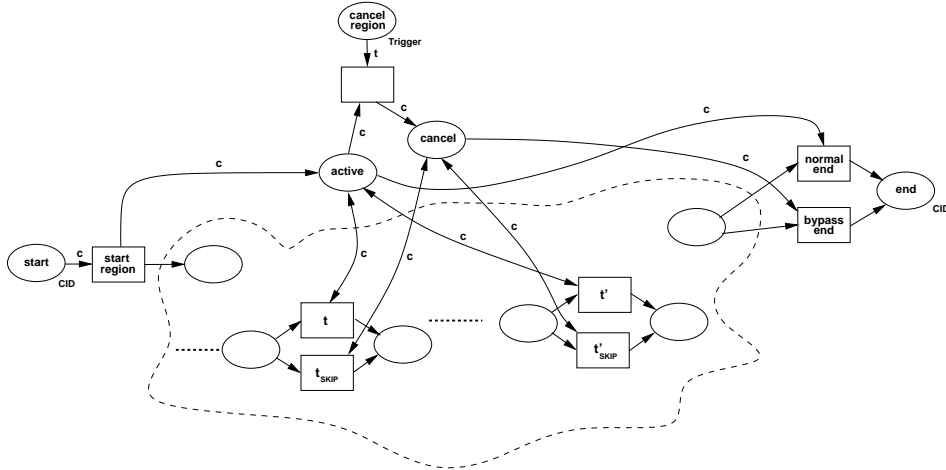


Figure 30: Cancel region implementation

action with an associated end event that allows all executing activities in a process instance to be terminated. Similarly UML 2.0 ADs achieve the same effect using the InterruptibleActivityRegion construct. FLOWer provide partial support for the pattern through its ability to skip or redo entire cases.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support for the pattern if it provides the ability to denote the cancellation of an entire process instance in a process model and satisfies the context requirements for the pattern. If there are any side-effects associated with the cancellation (e.g. forced completion of other activities, the process instance being marked as complete), then the offering is rated as having partial support.

## 4 New Control-Flow Patterns

Review of the patterns associated with the control-flow perspective over the past few years has led to the recognition that there are a number of distinct modelling constructs that can be identified during process modelling that are not adequately captured by the original set of twenty patterns. In this section, we present twenty three new control-flow patterns that augment the existing range of patterns described in the previous section and elsewhere [vdABtHK00, vdAtHKB03]. In an attempt to describe the operational characteristics of each pattern more rigourously, we also present a formal model in Coloured Petri-Net (CPN) format for each of them. In fact the explicit modelling of the original patterns using CPN Tools (cf. Section 3), helped identify a number of new patterns as well as delineating situations where some of the original patterns turned out to be collections of patterns.

When discussing the *Arbitrary Cycles* pattern (WCP-10), we indicated that some people refer to such cycles as “goto’s”. We disagree with this because if arbitrary “forward” graphical connections are allowed, then it does not make sense to forbid “backward” graphical connections on the basis that they constitute sloppy modeling. Nevertheless, it may be useful to have special constructs for structured loops as is illustrated by the next pattern.

### Pattern WCP-21 (Structured Loop)

**Description** The ability to execute an activity or sub-process repeatedly. The loop has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point.

#### Examples

- While the machine still has fuel remaining, continue with the production process.
- Only schedule flights if there is no storm activity.
- Continue processing photographs from the film until all of them have been printed.
- Repeat the *select player* activity until the entire team has been selected.

**Motivation** There are two general forms of this pattern – the *while* loop which equates to the classic `while...do` pre-test loop construct used in programming languages and the *repeat* loop which equates to the `repeat...until` post-test loop construct.

The while loop allows for the repeated sequential execution of a specified activity or a sub-process zero or more times providing a nominated condition evaluates to true. The pre-test condition is evaluated before the first iteration of the loop and is re-evaluated before each subsequent iteration. Once the pre-test condition evaluates to false, the thread of control passes to the activity immediately following the loop. The while loop structure ensures that each of the activities embodied within it are executed the same number of times.

The repeat loop allows for the execution of an activity or sub-process one or more times, continuing with execution until a nominated condition evaluates to true. The post-test condition is evaluated after the first iteration of the loop and is re-evaluated after each subsequent iteration. Once the post-test condition evaluates to true, the thread of control passes to the activity immediately following the loop. The repeat

loop structure ensures that each of the activities embodied within it are executed the same number of times.

**Context** As indicated above, there are two variants of this pattern: the while loop illustrated in Figure 31 and the repeat loop shown in Figure 32. In both cases, activity B is executed repeatedly.

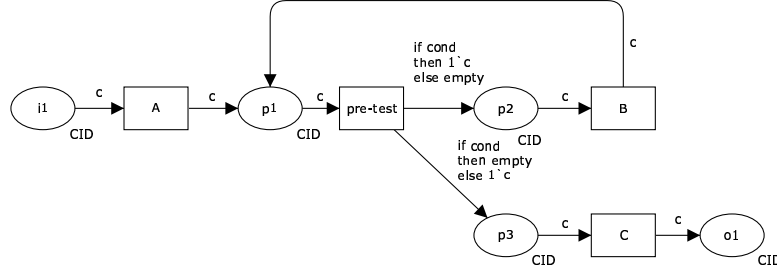


Figure 31: Structured loop pattern – while variant

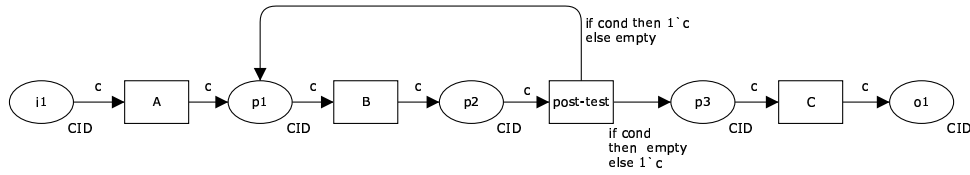


Figure 32: Structured loop pattern – repeat variant

**Implementation** The main consideration in supporting the *Structured Loop* pattern is the availability of a construct within a modelling language to denote the repeated execution of an activity or sub-process based on a specified condition. The evaluation of the condition to determine whether to continue (or cease) execution can occur either before or after the activity (or sub-process) has been initiated.

WebSphere MQ provides support for post-tested loops through the use of exit conditions on block or process constructs. Similarly, FLOWer provides the sequential plan construct that allows a sequence of activities to be repeated sequentially until a nominated condition is satisfied. iPlanet also supports post-tested loops through conditions on outgoing routers from an activity that loop back to the beginning of the same activity. BPEL directly supports pre-tested loops via the <while> construct. BPMN and XPDL allow both pre-tested and post-tested loops to be captured through the loop activity construct. Similarly UML 2.0 ADs provide the LoopNode construct which has similar capabilities. SAP provides two loop constructs corresponding to the while loop and the repeat loop. (In fact the SAP loop construct is more general merging both the while and repeat loop into a single construct).

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support for the pattern if it has a construct that denotes an activity or sub-process should be repeated whilst a specified condition remains true or until a specified condition becomes true.

Another new pattern related to loops is the *Recursion* pattern.

### Pattern WCP-22 (Recursion)

**Description** The ability of an activity to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated.

#### Example

- An instance of the *resolve-defect* activity is initiated for each mechanical problem that is identified in the production plant. During the execution of the *resolve-defect* activity, if a mechanical fault is identified during investigations that is not related to the current defect, another instance of the *resolve-defect* is started. These sub-processes can also initiate further *resolve-defect* activities should they be necessary. The parent *resolve-defect* activity cannot complete until all child *resolve-defect* activities that it initiated have been satisfactorily completed.

**Motivation** For some types of activity, particularly those that may involve unplanned repetition of an activity or sub-process, simpler and more succinct solutions can be provided through the use of recursion rather than iteration. In order to harness recursive forms of problem solving within the context of a workflow, a means of describing an activity execution in terms of itself (i.e. the ability for an activity to invoke another instance of itself whilst executing) are required.

**Context** Figure 33 illustrates the format of the recursion pattern in Petri-Net terms. Activity A can be decomposed into the process model with input *i1* and output *o1*. It is important to note that this process also contains the activity A hence the activity is described in terms of itself.

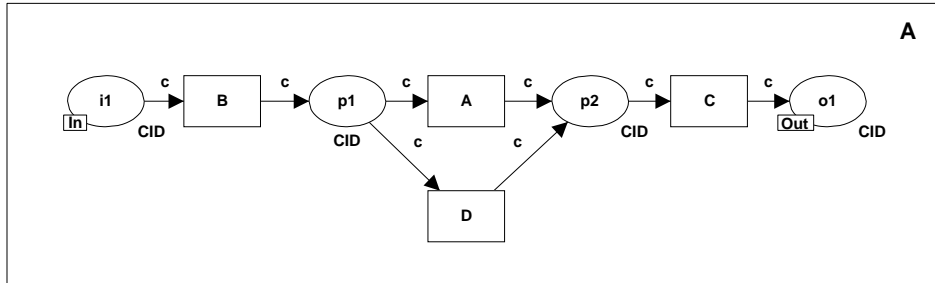


Figure 33: Recursion pattern

It is a context condition of this pattern that an offering provides the ability within a process model to denote the synchronous invocation of an activity or sub-process within the same process model. In order to ensure that use of recursion does not lead to infinite self-referencing decompositions, there is also a second context condition: there must be at least one path through the process decomposition which is not self-referencing and will terminate normally. In Figure 33, this is illustrated by the execution sequence BDC a token from input *i1* to output *o1*. This corresponds to the *terminating condition* in mathematical descriptions of recursion and ensures that where recursion is used in a process that the overall process will eventually complete normally when executed.

**Implementation** In order to implement recursion within the context of a workflow, some means of invoking a distinct instance of an activity is required from within a given activity implementation. Staffware, WebSphere MQ, COSA, iPlanet and SAP

Workflow all provide the ability for an activity to invoke an instance of itself whilst executing.

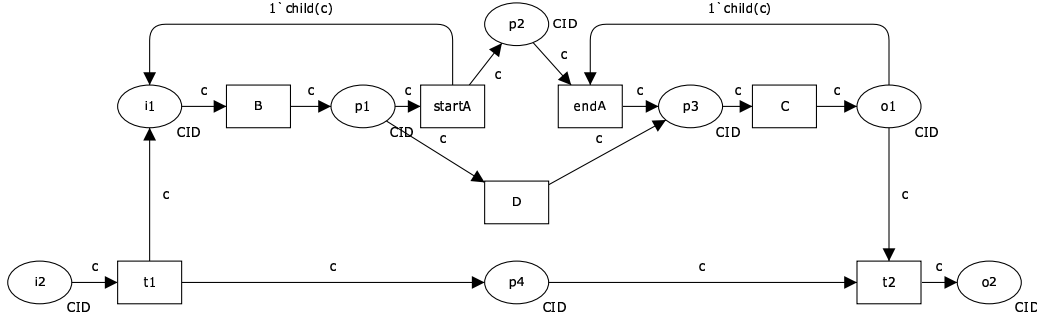


Figure 34: Recursion implementation

The actual mechanics of implementing recursion for a process such as that depicted in Figure 33 are shown in Figure 34. The execution of the recursive activity A is denoted by the transitions startA and endA. When an instance of activity A is initiated in a case c, any further execution of the case is suspended and the thread of control is passed to the decomposition that describes the recursive activity (in this case, activity B is enabled). A new case-id is created for the thread of control that is passed to the decomposition and a mapping function (in this example denoted by `child()`) is used to capture the relationship between the parent case-id and the decomposition case-id, thus ensuring that once the child case has completed, the parent case can continue from the point at which it originally suspended execution and invoked the child instance of itself.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it is able to satisfy the context criteria for the pattern.

The original patterns did not explicitly identify the notion of triggers. Here we identify two types of triggers: the *Transient Trigger* (WCP-23) and the *Persistent Trigger* (WCP-24) patterns.

### Pattern WCP-23 (Transient Trigger)

**Description** The ability for an activity to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving activity.

#### Examples

- Start the *Handle Overflow* activity immediately when the *dam capacity full* signal is received.
- If possible, initiate the *Check Sensor* activity each time an *alarm trigger signal* is received.

**Motivation** Transient triggers are a common means of signalling that a pre-defined event has occurred and that an appropriate handling response should be undertaken

– comprising either the initiation of a single activity, a sequence of activities or a new thread of execution in a process. Transient triggers are events which must be dealt with as soon as they are received. In other words, they must result in the immediate initiation of an activity. The workflow provides no form of memory for transient triggers. If they are not acted on immediately, they are irrevocably lost.

**Context** Transient triggers have two context conditions associated with them: (1) it must be possible to direct a trigger to a specific activity instance executing in a specific process instance and (2) if the activity instance to which the trigger is directed is not waiting (for the trigger) at the time that the trigger is received, then the trigger is lost. There are two main variants of this pattern depending on whether the process is executing in a safe execution environment or not. Figure 35 shows the safe variant, only one instance of activity B can wait on a trigger at any given time. Note that place p2 holds a token for any possible process instance. This place makes sure that at most one instance of activity B exists at any time.

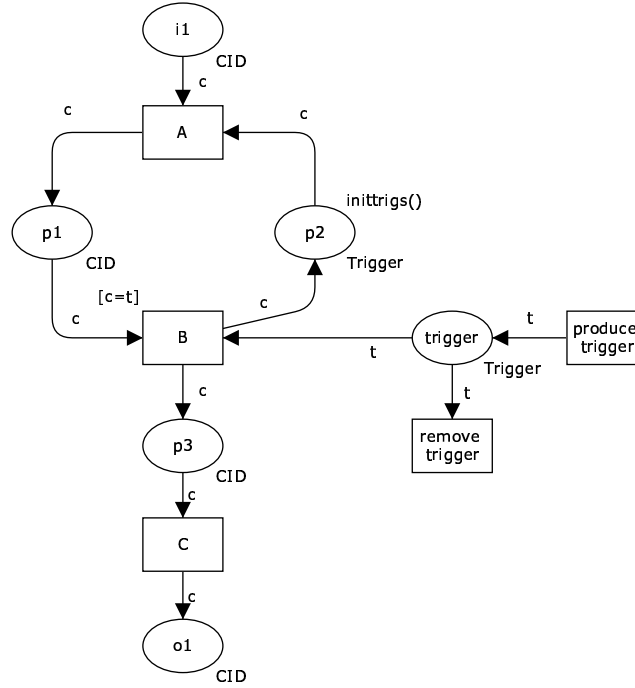


Figure 35: Transient trigger pattern

The alternative option for unsafe processes is shown in Figure 36. Multiple instances of activity B can remain waiting for a trigger to be received. However only one of these can be enabled for each trigger when it is received.

**Implementation** Staffware provides support for transient triggers via the Event Step construct. Similarly COSA provides the trigger construct which can operate in both synchronous and asynchronous mode supporting transient and persistent triggers respectively. Both of these offerings implement the safe form of the pattern (as illustrated in Figure 35). SAP Workflow provides similar support via the “wait for event” step construct. UML 2.0 ADs provide the ability for signals to be discarded where there are not immediately required through the explicit enablement feature of the AcceptEventAction construct which is responsible for handling incoming signals.

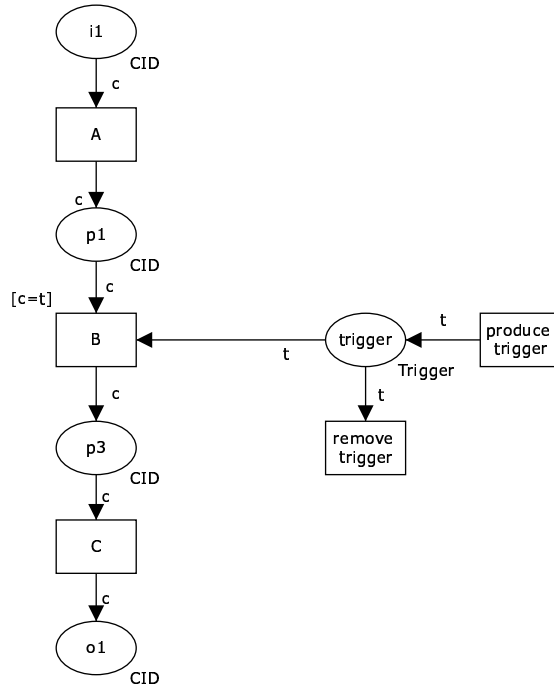


Figure 36: Transient trigger pattern (unsafe variant)

**Issues** One consideration that arises with the use of transient triggers is what happens when multiple triggers are received simultaneously or in a very short time interval. Are the latter triggers inherently lost as a trigger instance is already pending or are all instances preserved (albeit for a potentially short timeframe).

**Solutions** In general, in the implementations examined (Staffware, COSA and SAP Workflow) it seems that all transient triggers are lost if they are not immediately consumed. There is no provision for transient triggers to be duplicated.

**Evaluation Criteria** An offering achieves full support if it is able to satisfy the context criteria for the pattern.

### Pattern WCP-24 (Persistent Trigger)

**Description** The ability for an activity to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the workflow until they can be acted on by the receiving activity.

#### Examples

- Initiate the *Staff Induction* activity each time a *new staff member* event occurs.
- Start a new instance of the *Inspect Vehicle* activity for each *service overdue* signal that is received.

**Motivation** Persistent triggers are inherently durable in nature, ensuring that they are not lost in transit and are buffered until they can be dealt with by the target activity. This means that the signalling activity can be certain that the trigger will result in the activity to which they are directed being initiated either immediately (if it already has received the thread of control) or at some future time.

**Context** There are two variants of the persistent triggers. Figure 37 illustrates the situation where a trigger is buffered until control-flow passes to the activity to which the trigger is directed. Once this activity has received a trigger, it can commence execution. Alternatively, the trigger can initiate an activity (or the beginning of a thread of execution) that is not contingent on the completion of any preceding activities. This scenario is illustrated by Figure 38.

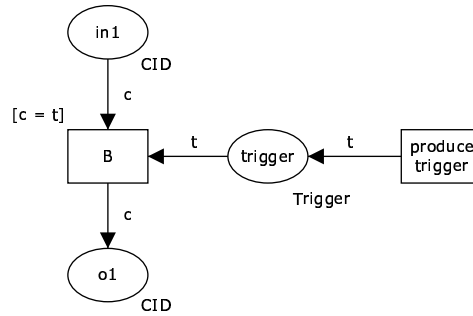


Figure 37: Persistent trigger pattern

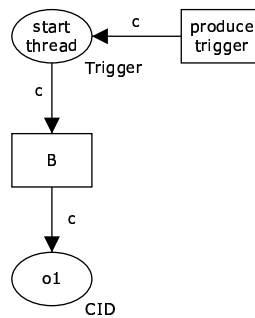


Figure 38: Persistent trigger pattern – new execution thread variant

**Implementation** Of the workflow systems examined, COSA provide support for persistent triggers via its integrated trigger construct, SAP Workflow has the “wait for event” step construct, FLOWer and FileNet provide the ability for activities to wait on specific data conditions that can be updated from outside the workflow. The business process modelling formalisms BPMN, XPDL and BPEL all provide a mechanism for this form of triggering via messages and in all cases the messages are assumed to be durable in nature and can either trigger a standalone activity or can enable a blocked activity waiting on receipt of a message to continue. UML 2.0 Activity Diagrams provides a similar facility using signals. Although EPCs provide support for multiple input events which can be utilized as persistent triggers, it is not possible to differentiate between them hence this is viewed as partial support.

Note that if the pattern is not directly supported, it is often possible to implement persistent triggers indirectly by adding a dummy activity which “catches” the trigger.

**Issues** None identified.

**Solutions** N/A



**Evaluation Criteria** An offering achieves full support for this pattern if it provides any form of durable activity triggering that can be initiated from outside the process environment. If triggers do not retain a discrete identity when received and/or stored, an offering is viewed as providing partial support.

Next we consider some additional patterns related to cancellation.

### Pattern WCP-25 (Cancel Region)

**Description** The ability to disable a set of activities in a process instance. If any of the activities are already executing, then they are withdrawn. The activities need not be a connected subset of the overall process model.

#### Examples

- Stop any activities in the *Prosecution* process which access the *evidence* database from running.
- Withdraw all activities in the *Waybill Booking* process after the *freight-lodgement* activity.

**Motivation** The option of being able to cancel a series of (potentially unrelated) activities is a useful capability, particularly for handling unexpected errors or for implementing forms of exception handling.

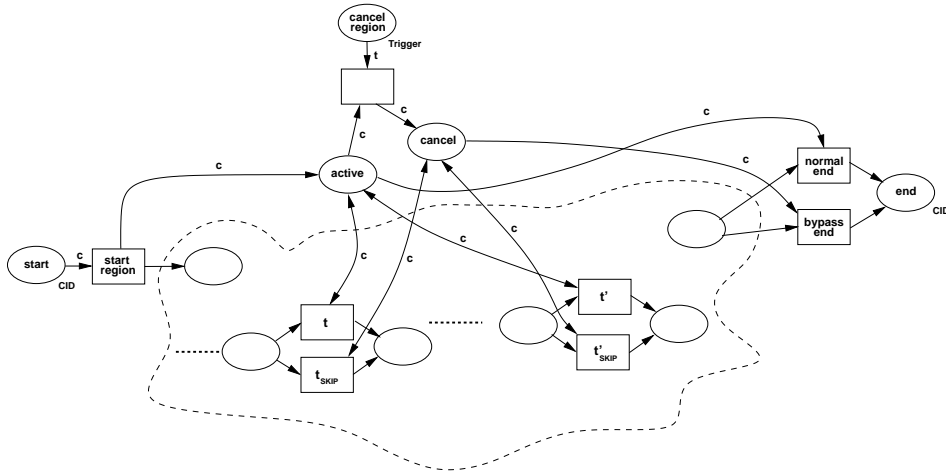


Figure 39: Cancel region implementation

**Context** The general form of this pattern is illustrated in Figure 39. It is based on the premise that every activity in the required region has an alternate “bypass” activity. When the cancellation of the region is required, the process instance continues execution, but the bypass activities are executed instead of the original activities. As a consequence, no further work occurs on the activities in the cancellation region. However, as shown for the *Cancel Case* (WCP-20) pattern, there are several alternative mechanisms that can be used to cancel parts of a process. There are two specific requirements for this pattern: (1) it must be possible to denote a set of (not necessarily connected) activities that are to be cancelled and (2) once cancellation of the region is invoked, all activity instances within the region (both currently executing and also those that may execute at some future time) must be withdrawn.

**Implementation** The concept of cancellation regions is not widely supported. Staffware offers the opportunity to withdraw steps but only if they have not already commenced execution. FLOWer allows individual activities to be skipped but there is no means of cancelling a group of activities. UML 2.0 Activity Diagrams are the only offering examined which provides complete support for this pattern: the *InterruptibleActivityRegion* construct allow a set of activities to be cancelled. BPMN and XPD L offer partial support by enclosing the activities that will potentially be cancelled in a sub-process and associating an error event with the sub-process to trigger cancellation when it is required. In both cases, the shortcoming of this approach is that the activities in the sub-process must be a connected subgraph of the overall process model. Similarly BPEL only supports cancellation of activities in the same scope hence it also achieves a partial rating. As COSA has an integrated notion of state, it is possible to implement cancellation regions in a similar way that presented in Figure 39 however the overall process model is likely to become intractable for cancellation regions of any reasonable scale hence this is viewed as partial support.

**Issues** One issue that can arise with the implementation of the *Cancel Region* pattern occurs when the cancelling activity lies within the cancellation region. Although this activity must run to completion and cause the cancellation of all of the activities in the defined cancellation region, once this has been completed, it too must be cancelled.

**Solutions** The most effective solution to this problem is to ensure that the cancelling activity is the last of those to be processed (i.e. the last to be terminated) of the activities in the cancellation region.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern.

### Pattern WCP-26 (Cancel Multiple Instance Activity)

**Description** Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. At any time, the multiple instance activity can be cancelled and any instances which have not completed are withdrawn. This does not affect activity instances that have already completed.

#### Example

- Run 500 instances of the *Protein Test* activity with distinct samples. If it has not completed one hour after commencement, cancel it.

**Motivation** This pattern provides a means of cancelling a multiple instance activity at any time during its execution such that any remaining instances are cancelled. However any instances which have already completed are unaffected by the cancellation.

**Context** There are two variants of this pattern depending on whether the activity instances are started sequentially or simultaneously. These scenarios are depicted in Figures 40 and 41. In both cases, transition C corresponds to the multiple instance activity, which is executed *numinst* times. When the cancel transition is enabled, any remaining instances of activity C that have not already executed are withdrawn, as is the ability to initiate any additional instances (via the **create instance** transition). No subsequent activities are enabled as a consequence of the cancellation. Note that both CPN models are assumed to operate in a safe context.

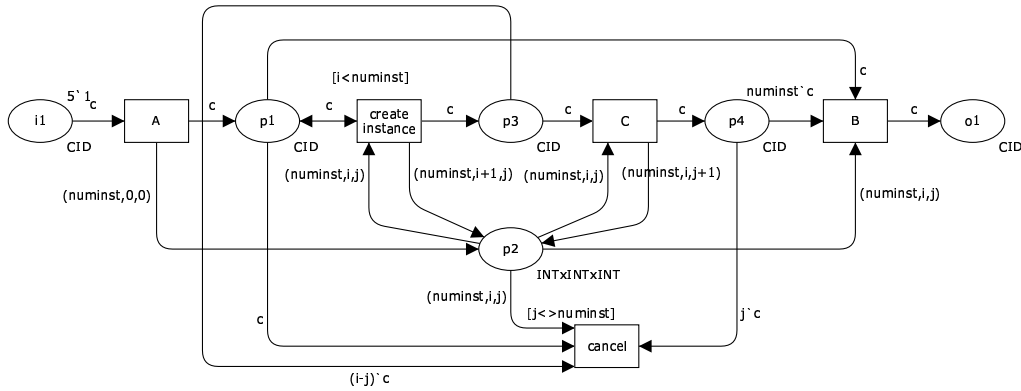


Figure 40: Cancel multiple instance activity pattern – sequential instances

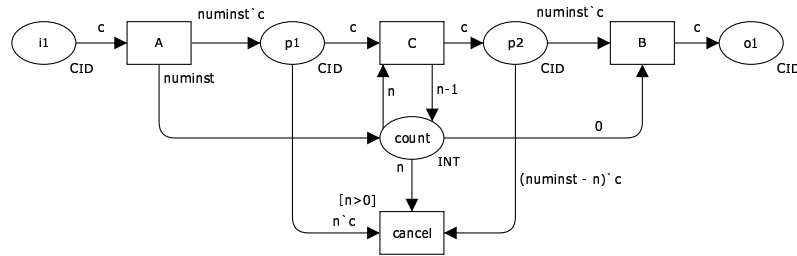


Figure 41: Cancel multiple instance activity pattern – concurrent instances

**Implementation** In order to implement this pattern, an offering also needs to support one of the Multiple Instance patterns that provide synchronization of the activity instances at completion (i.e. WCP-13 – WCP-15). Staffware provides the ability to immediately terminate dynamic subprocedures albeit with loss of any associated data. SAP Workflow allows multiple instances created from a “multi-line container element” to be terminated when the parent activity terminates. BPMN and XPDL support the pattern via a MI task which has an error type intermediate event trigger at the boundary. When the MI activity is to be cancelled, a cancel event is triggered to terminate any remaining MI activities. Similarly UML 2.0 ADs provide support by including the multiple instance activity in a cancellation region. Oracle BPEL is able to support the pattern by associating a fault or compensation handler with a <flowN> construct.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. If there are any limitations on the range of activities that can appear within the cancellation region or the types of activity instances that can be cancelled then an offering achieves a partial rating.

### Pattern WCP-27 (Complete Multiple Instance Activity)

**Description** Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize

the instances at completion before any subsequent activities can be triggered. During the course of execution, it is possible that the activity needs to be forcibly completed such that any remaining instances are withdrawn and the thread of control is passed to subsequent activities.

### Example

- Run 500 instances of the *Protein Test* activity with distinct samples. One hour after commencement, withdraw all remaining instances and initiate the next activity.

**Motivation** This pattern provides a means of finalising a multiple instance activity that has not yet completed at any time during its execution such that any remaining instances are withdrawn and the thread of control is immediately passed to subsequent activities. Any instances which have already completed are unaffected by the cancellation.

**Context** There are two variants of this pattern depending on whether the activity instances are started sequentially or simultaneously. These scenarios are depicted in Figures 42 and 43. In both cases, transition C corresponds to the multiple instance activity, which is executed  $numinst$  times. When the cancel transition is enabled, any remaining instances of activity C that have not already executed are withdrawn, as is the ability to add any additional instances (via the add transition). The subsequent activity (illustrated by transition B) is enabled immediately. Note that both CPN models are assumed operate in a safe context.

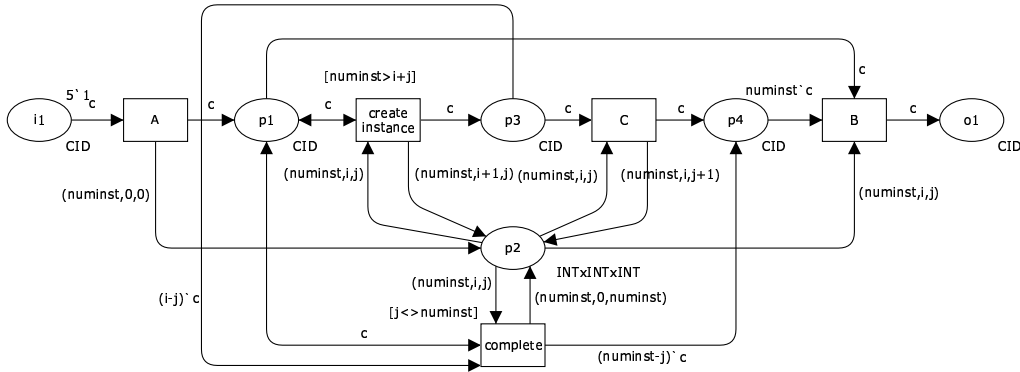


Figure 42: Complete multiple instance activity pattern – sequential instances

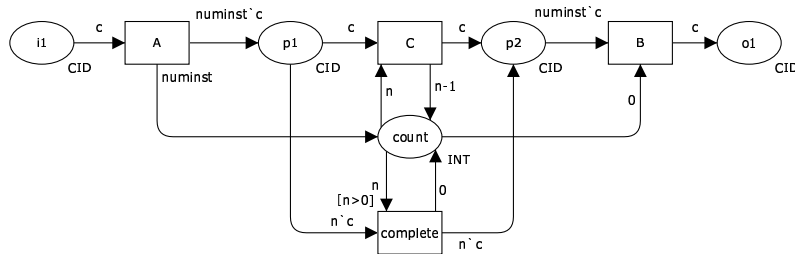


Figure 43: Complete multiple instance activity pattern – concurrent instances

**Implementation** In order to implement this pattern, an offering also needs to support one of the Multiple Instance patterns that provide synchronization of the activity instances at completion (i.e. WCP-13 – WCP-15). FLOWer provides indirect

support for this pattern via the auto-complete condition on dynamic plans which force-completes unfinished plans when the condition evaluates to true however this can only occur when all subplans have completed. Similarly, it also provides deadline support for dynamic plans which ensures that all remaining instances are forced complete once the deadline is reached, however this action also causes all subsequent activities to be force completed as well.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. It demonstrates partial support if there are limitations on when the completion activity can be initiated or if the force completion of the remaining instances does not result in subsequent activities in the process instance being triggered normally.

The *Structured Discriminator* (WCP-9) pattern is assumed to operate in a safe context and waits for the completion of all branches before it resets. Therefore, we propose two extensions of the basic pattern which relax some of these context assumptions and allow it to operate in different scenarios. These variants are the *Blocking Discriminator* and the *Cancelling Discriminator*.

### Pattern WCP-28 (Blocking Discriminator)

**Description** The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The discriminator construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the discriminator has reset.

#### Example

- When the first member of the *visiting delegation* arrives, the *check credentials* activity can commence. It concludes when all delegation members have arrived. Owing to staff constraints, only one instance of the *check credentials* activity can be undertaken at any time. Should members of another delegation arrive, the checking of their credentials is delayed until the first *check credentials* activity has completed.

**Motivation** The *Blocking Discriminator* pattern is a variant of the *Structured Discriminator* pattern that is able to run in environments where there are potentially several concurrent execution threads within the same process instance. This quality allows it to be used in loops and other process structures where more than one execution thread may be received in a given branch in the time between the first branch being enabled and the *Discriminator* being reset.

**Context** Figure 44 illustrates the operation of this pattern. It is more robust than the *Structured Discriminator* as it is not subject to the constraint that each incoming branch can only being triggered once prior to reset. However it does have the context condition that the *Discriminator* construct can only deal with one case at a time (i.e. once one of the incoming places *i1* to *im* is triggered for a given case, all other incoming triggers that are received are expected to relate to the same case).

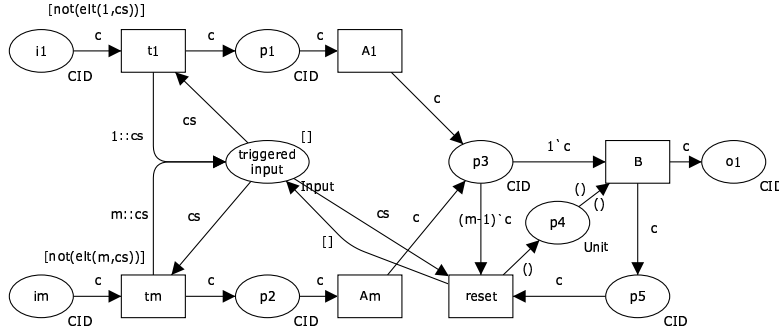


Figure 44: Blocking discriminator pattern

The *Blocking Discriminator* functions by keeping track of which inputs have been triggered (via the **triggered input** place) and preventing them from being re-enabled until the construct has reset as a consequence of receiving a trigger on each incoming place. An important feature of this pattern is that it is able to be utilized in environments that do not support a safe process model or those that may receive multiple triggerings on the same input place e.g. where the *Discriminator* is used within a loop.

A variation to this process model where the **triggered input** place is extended to keep track of both case-id and input branch is shown in Figure 45. This allows the *Discriminator* to operate in a concurrent environment where it may need to handle multiple cases simultaneously<sup>6</sup>.

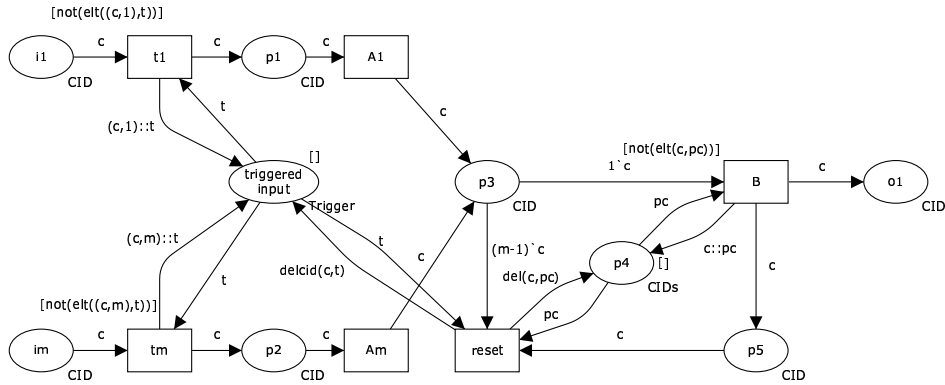


Figure 45: Blocking discriminator pattern – extension for concurrent process instances

**Implementation** In the event of concurrent process instances attempting to simultaneously initiate the same *Discriminator*, it is necessary to keep track of both the process instance and the input branches that have triggered the *Discriminator* and also the execution threads that are consequently blocked until it completes. The *Blocking Discriminator* is partially supported by BPMN, XPDL and UML 2.0 ADs .

**Issues** None identified.

<sup>6</sup>In many of the diagrams we assume that the context is restricted to one process instance executing in isolation. This is the case in Figure 44 where we do not distinguish different process instances (i.e., case) in the list of input already received. Note that the type of generalization applied in Figure 45 could also be applied to earlier diagrams.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. If there is any ambiguity in how the join condition is specified, an offering is considered to provide partial support for the pattern.

### Pattern WCP-29 (Cancelling Discriminator)

**Description** The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the discriminator also cancels the execution of all of the other incoming branches and resets the construct.

### Example

- After the *extract-sample* activity has completed, parts of the sample are sent to three distinct laboratories for examination. Once the first of these laboratories completes the *sample-analysis*, the other two activity instances are cancelled and the *review-drilling* activity commences.

**Motivation** This pattern provides a means of expediting a process instance where a series of incoming branches to a join need to be synchronized but it is not important that the activities associated with each of the branches (other than the first of them) be completed.

**Context** The operation of this pattern is shown in Figure 46. It is a context condition of this pattern that only one thread of execution is active for a given process instance in each of the preceding branches to the *Discriminator* prior to it being reset. If this is not the case, then the behaviour of the process instance is likely to become unpredictable at a later stage during execution.

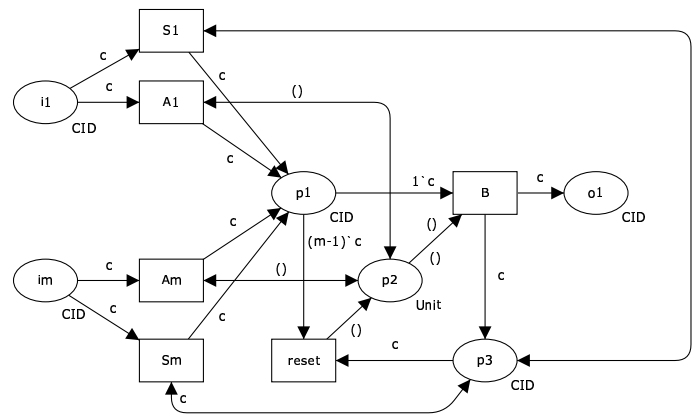


Figure 46: Cancelling discriminator pattern

Inputs **i1** to **im** to the *Discriminator* serve to identify the branches preceding the construct. Transitions **A1** to **Am** signify activities in these preceding branches. Transitions **S1** to **Sm** indicate alternate “bypass” or “cancellation” activities for each of these branches (these execution options are not initially available to incoming execution threads). The first control-flow token for a given case received at any input will

cause B to fire and put a token in *o1*. As soon as this occurs, subsequent execution threads on other branches are put into “bypass mode” and instead of executing the normal activities (*A1*..*Am*) on their specific branch, they can execute the “bypass” transitions (*S1*..*Sm*). (Note that the bypass transitions do not require any interaction. Hence they are executed directly by the process engine and we can assume that the skip transitions are executed once they are enabled and complete almost instantaneously hence expediting completion of the branch). Once all incoming branches for a given case have been completed, the *Discriminator* construct can then reset and be re-enabled again for the same case.

As with the *Blocking Discriminator* pattern, there is a variation (illustrated in Figure 47) that enables the *Discriminator* to function in highly concurrent environments where multiple process instances may need to be processed by the *Discriminator* simultaneously. This is achieved by extending place *p2* to keep track of process instances which have triggered the *Discriminator* but not yet caused it to reset.

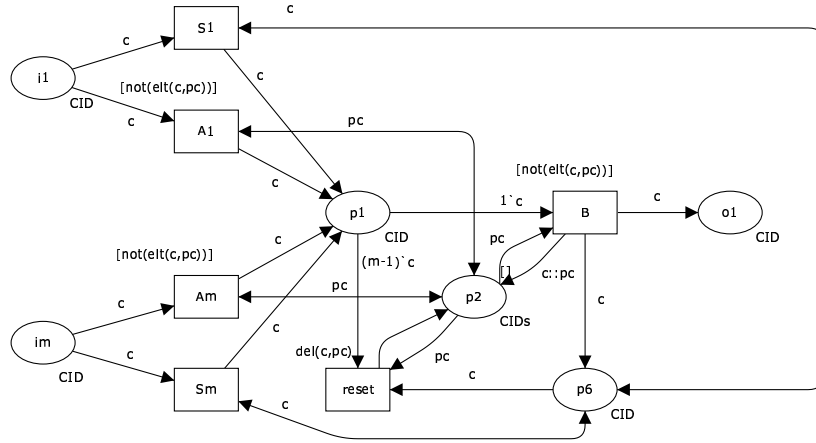


Figure 47: Cancelling discriminator pattern – extension for concurrent process instances

**Implementation** In order to implement this pattern, it is necessary for the offering to support some means of denoting the extent of the incoming branches to be cancelled. This can be based on the *Cancel Region* pattern although support is only required for a restricted form of the pattern as the region to be cancelled will always be a connected subgraph of the overall process model with the *Discriminator* construct being the connection point for all of the incoming branches.

This pattern is supported by the fork construct in SAP Workflow with the number of branches required for completion set to one. In BPMN it is achieved by incorporating the incoming branches and the discriminator in a sub-process that has an error event associated with it. The error event is triggered, cancelling the remaining branches in the sub-process, when the *Discriminator* is triggered by first incoming branch. This configuration is illustrated in Figure 48(a). A similar solution is available in XPD. UML 2.0 ADs support the pattern in a similar way by enclosing all of the incoming branches in an InterruptibleActivityRegion which is cancelled when the *Discriminator* fires.

**Issues** The major difficulty with this pattern is in determining how much of the process model preceding the *Discriminator* is to be included in the cancellation region.



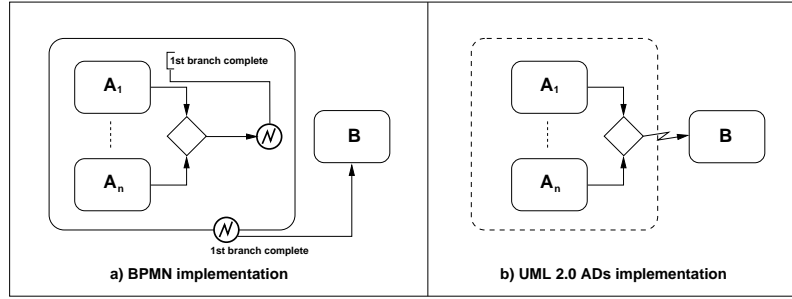


Figure 48: Cancelling discriminator pattern implemented in BPMN and UML 2.0 ADs

**Solutions** This issue is easily addressed in structured workflows as all of the branches back to the preceding split construct which corresponds to the *Discriminator* should be subject to cancellation. In Figure 49(a), it is easy to see that the area denoted by the dotted box should be the cancellation region. It is a more complex matter when the workflow is not structured as in Figure 49(b) or other input arcs exist into the preceding branches to the *Discriminator* that are not related to the corresponding split as shown in Figure 49(c). In both of these situations, the overall structure of the process leading up to the *Discriminator* serves as a determinant of whether the pattern can be supported or not. In Figure 49(b), a cancellation region can be conceived which reaches back to the first AND-split and the pattern can be implemented based on this. A formal approach to determining the scope of the cancellation region can be found in [vdA01]. In Figure 49(c), the potential for other control-flows to be introduced which do not relate to the earlier AND-split, means that the pattern probably cannot be supported in a process model of this form.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. An offering is considered to provide partial support for the pattern if there are side-effects associated with the execution of the pattern (e.g. activities in incoming branches which have not completed being recorded as complete).

As discussed before, the *Partial Join* can be seen as a generalization of *Discriminator* pattern (i.e., a 1-out-of-M join). Hence, we introduce some patterns generalizing the different variants of the *Discriminator* pattern (i.e. where  $N > 1$ ).

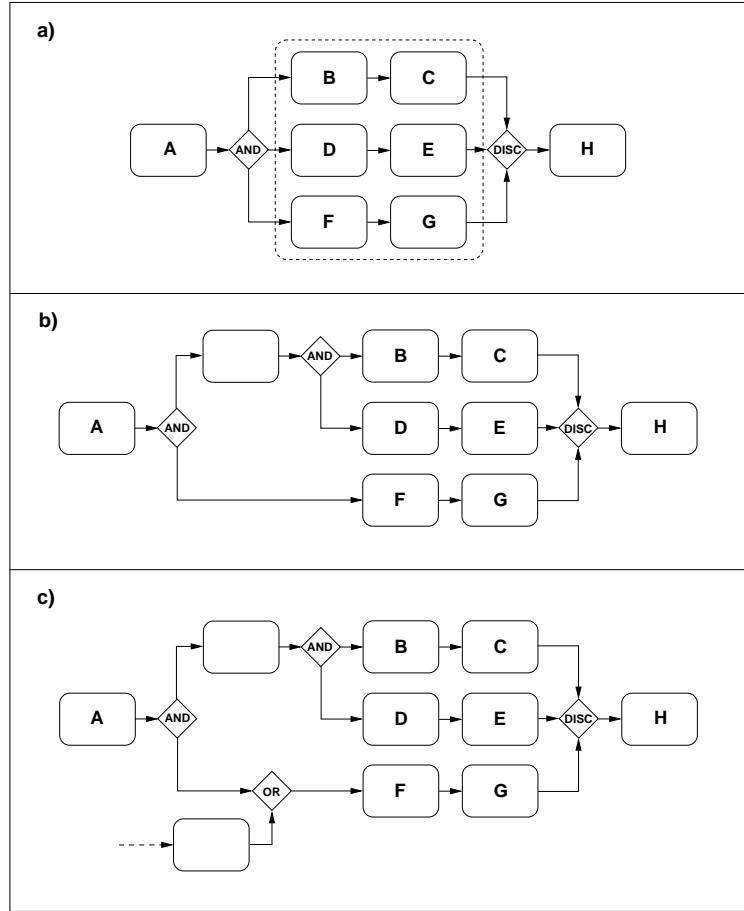


Figure 49: Process structure considerations for cancelling discriminator

### Pattern WCP-30 (Structured Partial Join)

**Description** The convergence of M branches into a single subsequent branch following a corresponding divergence earlier in the process model. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled.

#### Example

- Once two of the preceding three *Expenditure Approval* activities have completed, trigger the *Issue Cheque* activity. Wait until the remaining activities have completed before allowing the *Issue Cheque* activity to fire again.

**Motivation** The *Structured Partial Join* pattern provides a means of merging two or more distinct branches resulting from a specific parallel split or AND-split construct earlier in a workflow process into a single branch. The join construct does not require triggers on all incoming branches before it can fire. Instead a given threshold can be defined which describes the circumstances under which the join should fire – typically this is presented as the ratio of incoming branches that need to be live for firing as against the total number of incoming branches to the join e.g. a 2-out-of-3 Join

signifies that the join construct should fire when two of three incoming arcs are live. Subsequent completions of other remaining incoming branches have no effect on (and do not trigger) the subsequent branch. As such, the *Structured Partial Join* provides a mechanism for progressing the execution of a process once a specified number of concurrent activities have completed rather than waiting for all of them to complete.

**Context** The *Structured Partial Join* pattern is one possible variant of the AND-Join construct where the number of incoming arcs that will cause the join to fire ( $N$ ) is between 2 and  $M - 1$  (i.e. the total number of incoming branches less one i.e.  $2 \leq N < M$ ). There are a number of possible specializations of the AND-join pattern and they form a hierarchy based on the value of  $N$ . Where only one incoming arc must be live for firing (i.e.  $N=1$ ), this corresponds to one of the variants of the *Discriminator* pattern (cf. WCP-9, WCP-28 and WCP-29). An AND-Join where all incoming arcs must be live (i.e.  $N=M$ ) is the *Synchronization* or *Generalized AND-Join* pattern (WCP-3 or WCP-33) which is described below. There are a number of relationships between control-flow patterns and this issue is discussed in more depth in Section 6.

The pattern provides a means of merging two or more branches in a workflow and progressing execution of the workflow as rapidly as possible by enabling the subsequent (merged) branch as soon as a thread of control has been received on  $N$  of the incoming branches where  $N$  is less than the total number of incoming branches. There are two context conditions associated with the use of this pattern:

1. Each of the incoming branches to the join must only be triggered once prior to it being reset; and
2. The *Partial Join* resets (and can be re-enabled) once all of its incoming branches have been enabled precisely once.

The semantics of the *Structured Partial Join* pattern are illustrated in Figure 50. Note that  $B$  requires  $n$  tokens in place  $p1$  to progress.

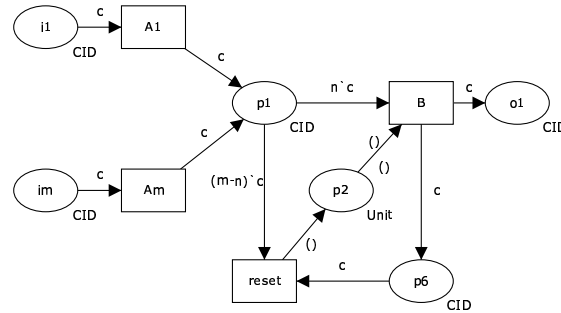


Figure 50: Structured partial join pattern

There are two possible variants on this pattern that arise from relaxing some of the context conditions associated with it. Both of these improve on the efficiency of the join whilst retaining its overall behaviour. The first alternative, the *Blocking Partial Join* (WCP-31) removes the requirement that each incoming branch can only be enabled once between join resets. It allows each incoming branch to be triggered multiple times although the construct only resets when one triggering has been received on each input branch. It is illustrated in Figure 51 and discussed in further

detail on page 61. Second, the *Cancelling Partial Join* (WCP-32), improves the efficiency of the pattern further by cancelling the other incoming branches to the join construct once N incoming branches have completed. It is illustrated in Figure 53 and discussed in further detail on page 62.

**Implementation** One of the difficulties in implementing the *Partial Join* is that it essentially requires a specific construct to represent the join if it is to be done in a tractable manner. iPlanet does so via the router construct which links preceding activities to a target activity. A router can have a custom trigger condition specified for it that causes the target activity to trigger when N incoming branches are live. SAP Workflow provides partial support for this pattern via the fork construct although any unfinished branches are cancelled once the first completes. None of the other workflow systems examined offers a dedicated construct. Staffware provides for a 1-out-of-2 join, but more complex joins must be constructed from this resulting in an over-complex process model. Similar difficulties exist for COSA. Of the business modelling languages, both BPMN and XPDL appear to provide support for the *Partial Join* via the complex gateway construct but the lack of detail on how the IncomingCondition results in a partial rating. UML 2.0 ADs also suffers from a similar lack of detail on the JoinSpec configuration required to support this pattern. There is no ability to represent the construct in BPEL.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. If there is any ambiguity in how the join condition is specified, an offering is considered to provide partial support for the pattern.

### Pattern WCP-31 (Blocking Partial Join)

**Description** The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled. The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset.

#### Example

- When the first member of the *visiting delegation* arrives, the *check credentials* activity can commence. It concludes when 80% of delegation members have arrived. Owing to staff constraints, only one instance of the *check credentials* activity can be undertaken at any time. Should members of another delegation arrive, the checking of their credentials is delayed until the first *check credentials* activity has completed.

**Motivation** The *Blocking Partial Join* is a variant of the *Structured Partial Join* that is able to run in environments where there are concurrent process instances, particularly process instances that have multiple concurrent execution threads.

**Context** Figure 51 illustrates the operation of this pattern. The *Blocking Partial Join* functions by keeping track of which inputs have been enabled (via the **triggered input** place) and preventing them from being re-enabled until the construct has reset

as a consequence of receiving a trigger on each incoming place. After  $N$  incoming triggers have been received for a given process instance (via tokens being received in  $N$  distinct input places from  $i1$  to  $im$ ), the join fires and a token is placed in output  $o1$ . The completion of the remaining  $N-M$  branches has no impact on the join except that it is reset when the last of them is received.

The pattern shares the same advantages over the *Structured Partial Join* as the *Blocking Discriminator* does over the *Structured Discriminator*, namely greater flexibility as it is able to deal with the situation where a branch is triggered more than once e.g. where the construct exists within a loop. It also shares the same context condition: it can only deal with one case at a time (i.e. once one of the incoming places  $i1$  to  $in$  is triggered for a given case, all other incoming triggers that are received are assumed to relate to the same case).

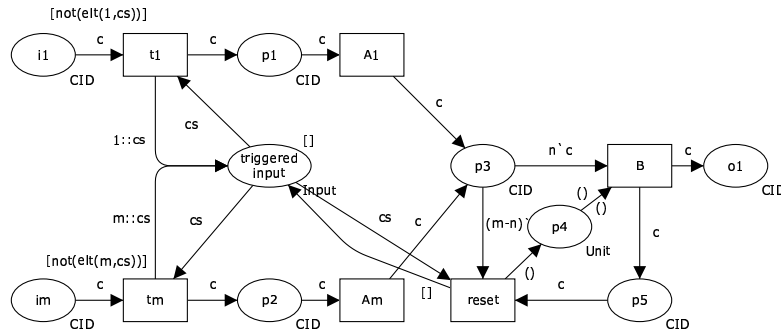


Figure 51: Blocking partial join pattern

There is a variation to this process model where the **triggered input** place is extended to keep track of both case and enabled input branches and place  $p4$  is extended to keep track of cases where the join has triggered (but not yet reset) is shown in Figure 52. This allows the join to operate in a concurrent environment where it may need to handle multiple cases simultaneously<sup>7</sup>.

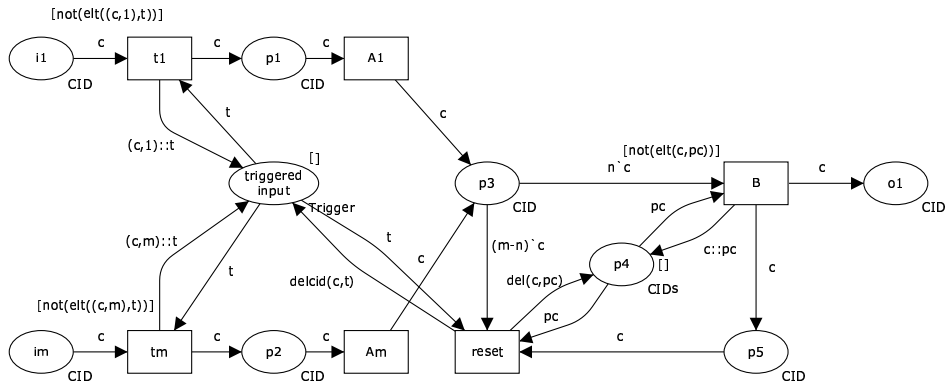


Figure 52: Blocking partial join pattern – extension for concurrent process instances

<sup>7</sup>This is a similar analogy to that noted for the *Blocking Discriminator* pattern. Figure 51 assumes that only one process instance executes in isolation where as Figure 52 is capable of distinguishing between and managing instances of distinct processes.

**Implementation** The approach to implementing this pattern is essentially the same as that for the *Blocking Discriminator* except that the join fires when N incoming branches have triggered rather than just the first. The *Blocking Partial Join* is partially supported by BPMN, XPDL and UML 2.0 ADs.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. If there is any ambiguity in how the join condition is specified, an offering is considered to provide partial support for the pattern.

### Pattern WCP-32 (Cancelling Partial Join)

**Description** The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled. Triggering the join also cancels the execution of all of the other incoming branches and resets the construct.

#### Example

- Once the picture is received, it is sent to three art dealers for the examination. Once two of the *prepare condition report* activities have been completed, the remaining *prepare condition report* activity is cancelled and the *plan restoration* activity commences.

**Motivation** This pattern provides a means of expediting a process instance where a series of incoming branches to a join need to be synchronized but only a subset of those activities associated with each of the branches needs to be completed.

**Context** The operation of this pattern is shown in Figure 53. It is a context condition of this pattern that only one thread of execution is active for a given process instance in each of the preceding branches to the discriminator. If this is not the case, then the behaviour of the process instance is likely to become unpredictable at a later stage during execution.

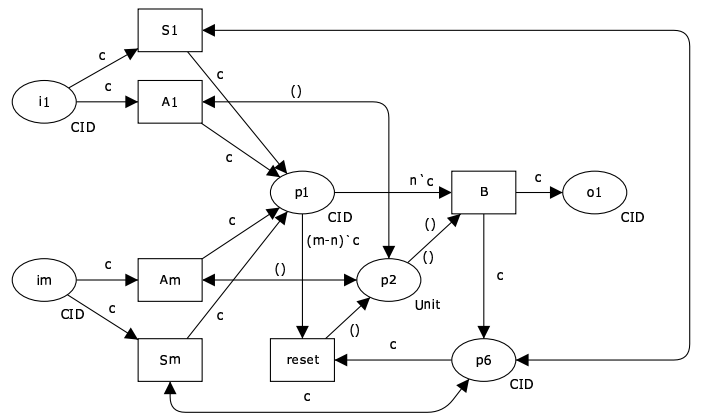


Figure 53: Cancelling partial join pattern

As with the *Cancelling Discriminator* pattern, there is a variation (illustrated in Figure 54) that enables the join to function in concurrent environments where multiple process instances may need to be processed by the join construct simultaneously. This is achieved by extending the p2 place to keep track of cases where the join has been triggered but not yet reset.

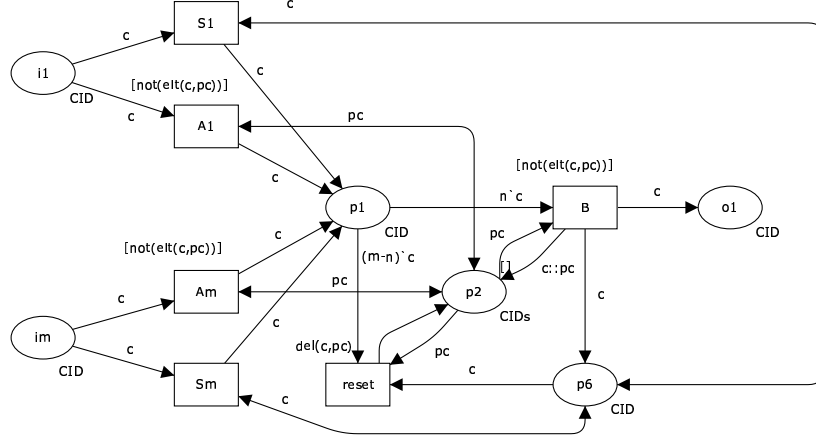


Figure 54: Cancelling partial join pattern – extension for concurrent process instances

**Implementation** The approach to implementing this pattern is essentially the same as that for the *Cancelling Discriminator* except that the join fires when N incoming branches have triggered rather than just the first. The *Cancelling Partial Join* is supported by SAP Workflow and UML 2.0 ADs. BPMN and XPDl achieve a partial support rating as it is unclear exactly how the join condition is specified.

**Issues** As for the *Cancelling Discriminator* pattern.

**Solutions** As for the *Cancelling Discriminator* pattern.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. An offering is considered to provide partial support for the pattern if there are undesirable side-effects associated with the construct firing (e.g. activities in incoming branches which have not completed being recorded as complete) or if the semantics associated with the join condition are unclear.

Many of the advanced synchronization patterns assume a safe context (i.e. a place cannot be marked twice for the same process instance). The following pattern is not predicated on this assumption and corresponds exactly to a transition in a non-safe Petri net.

### Pattern WCP-33 (Generalized AND-Join)

**Description** The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings.

## Examples

- When all *Get Directors Signature* activities have completed, run the *Complete Contract* activity.
- Accumulate engine, chassis and body components from the various production lines. When one of each has been received, use one of each component to assemble the basic car.

**Motivation** The *Generalized AND-Join* corresponds to one of the generally accepted notions of an AND-join implementation (the other situation is described by the *Synchronization* pattern) in which several paths of execution are synchronized and merged together. Unlike the *Synchronization* pattern, it supports the situation where one or more incoming branches may receive multiple triggers for the same process instance (i.e. a non-safe context).

**Context** The operation of the *Generalized AND-Join* is illustrated in Figure 55. Before transition A can be enabled, an input token (corresponding to the same case) is required in each of the incoming places (i.e. i1 to i3). When there are corresponding tokens in each place, transition A is enabled and consumes a token from each input place and once it has completed, deposits a token in output place o1. If there is more than one token at an input place, it ignores additional tokens and they are left in place.

The process analogy to this sequence of events is that the AND-join only fires when a trigger has been received on each incoming branch for a given process instance however additional triggers are retained for future firings. This approach to AND-join implementation relaxes the context condition associated with the *Synchronization* pattern that only allows it to receive one trigger on each incoming branch and as a result, it is able to be used in concurrent execution environments such as process models which involve loops as well as offerings that do not assume a safe execution environment.

One consideration associated with the *Generalized AND-Join* is that over time, each of the incoming branches should deliver the same number of triggers to the AND-join construct (although obviously, the timing of these triggers may vary). If this is not the case, then there is the potential for deadlocks to occur and/or tokens to remain after execution has completed.

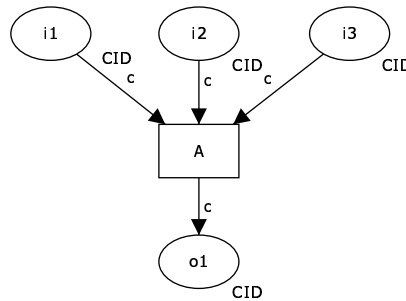


Figure 55: Generalised AND-join pattern

**Implementation** This need to provide persistence of triggerings (potentially between distinct firings of the join) means that this construct is not widely supported by



the workflow engines and business process modelling languages examined and only FileNet provides a construct for it. Token-based process models such as BPMN and XPD L have an advantage in this regard and both modelling notations are able to support this pattern<sup>8</sup>. EPCs provide a degree of ambiguity in their support for this pattern – whilst most documentation indicates that they do not support it, in the ARIS Simulator, they exhibit the required behaviour – hence they are awarded a partial support rating on account of this variance.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. If there is any ambiguity associated with the specification or use of the construct, an offering is considered to provide partial support for the pattern.

The multiple instance patterns considered earlier are based on the assumption that subsequent activities should be triggered only when all instances have completed. The following three patterns provide for a partial (i.e. an N-out-of-M) join between instances thus allowing subsequent activities to be triggered once a threshold of concurrent activities has been reached.

#### **Pattern WCP-34 (Static Partial Join for Multiple Instances)**

**Description** Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances is known when the first activity instance commences. Once N of the activity instances have completed, the next activity in the process is triggered. Subsequent completions of the remaining M-N instances are inconsequential.

#### **Example**

- Examine 10 samples from the production line for defects. Continue with the next activity when 7 of these examinations have been completed.

**Motivation** The *Static Partial Join for Multiple Instances* pattern is an extension to the *Multiple Instances with a priori Runtime Knowledge* pattern which allows the process instance to continue once a given number of the activity instances have completed rather than requiring all of them to finish before the subsequent activity can be triggered.

**Context** The general format of the *Static Partial Join for Multiple Instances* pattern is illustrated in Figure 56. Transition A corresponds to the multiple instance activity. There are several context conditions associated with this pattern:

- The number of concurrent activity instances (denoted by variable *m* in Figure 56) is known prior to activity commencement;
- The number of activities that need to be completed before subsequent activities in the process model can be triggered is known prior to activity commencement. This is denoted by variable *n* in Figure 56;

---

<sup>8</sup>Although we note that these formalisms are modelling languages which do not need to specify how a given construct will actually be realized

- Once the required number of activities have completed, the thread of control can immediately be passed to subsequent activities;
- The number of instances that must complete for the join to be triggered ( $n$ ) cannot be greater than the total number of concurrent activity instances ( $m$ ), i.e.  $n \not> m$ ; and
- Completion of the remaining activity instances do not trigger a subsequent activity, however all instances must have completed in order for the join construct to reset and be subsequently re-enabled.

In terms of the operation of this pattern, once the input place **i1** is triggered for a case,  $m$  instances of the multi-instance activity **A** are initiated concurrently and a “active” status is recorded for the pattern. These instances proceed independently and once  $n$  of them have completed, the join can be triggered and a token placed in output place **o1** signalling that the thread of control can be passed to subsequent activities in the process model. Simultaneously with the join firing, the token is removed from the **active** place allowing the remaining  $n - m$  activities complete. Once all  $m$  instances of activity **A** have finished, the status of the pattern changes to “ready” allowing it to be re-enabled.

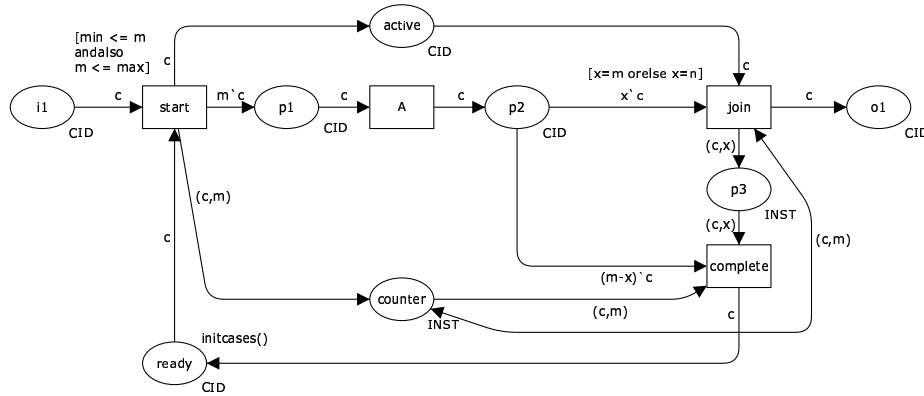


Figure 56: Static partial join implementation for multiple instances

There are two variants of this pattern which allow for some of the context conditions described above to be relaxed. First, the *Cancelling Partial Join for Multiple Instances* pattern removes the last context constraint by cancelling any remaining activity instances as soon as the join fires. It is illustrated in Figure 57 and discussed further on page 67.

The second, the *Dynamic Partial Join for Multiple Instances* pattern relaxes the first context condition and allows the value of  $m$  to be determined during the execution of the activity instances. In particular, it allows additional activity instances to be created “on the fly”. This pattern is illustrated in Figure 58 and described in further detail on page 69.

**Implementation** BPMN and XPDL both appear to offer support for this pattern via the Multiple Instance Loop Activity construct where the MI Flow\_Condition attribute is set to complex and ComplexMIFlowCondition is an expression that evaluates to true when exactly  $M$  instances have completed causing a single token to be passed on to the following activity. However no detail is provided to explain

how the ComplexMLFlowCondition is specified hence this is considered to constitute partial support for the pattern.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern. It achieves partial support if there is any ambiguity associated with the specification of the join condition.

### Pattern WCP-35 (Cancelling Partial Join for Multiple Instances)

**Description** Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances is known when the first activity instance commences. Once N of the activity instances have completed, the next activity in the process is triggered and the remaining M-N instances are cancelled.

#### Example

- Run 500 instances of the *Protein Test* activity with distinct samples. Once 400 of these have completed, cancel the remaining instances and initiate the next activity.

**Motivation** This pattern provides a variant of the multiple instances pattern that expedites process throughput by both allowing the process to continue to the next activity once a specified number (N) of the multiple instance activities have completed and also cancels any remaining activity instances negating the need to expend any further effort executing them.

**Context** Figure 57 illustrates the operation of this pattern. It is similar in form to that for the *Static Partial Join for Multiple Instances* pattern (WCP-34) but functions in a different way once the join has fired. At this point any remaining instances which have not already commenced are “bypassed” by allowing the **skip** activity to execute in their place. The **skip** activity executes almost instantaneously for those and the pattern is almost immediately able to reset.

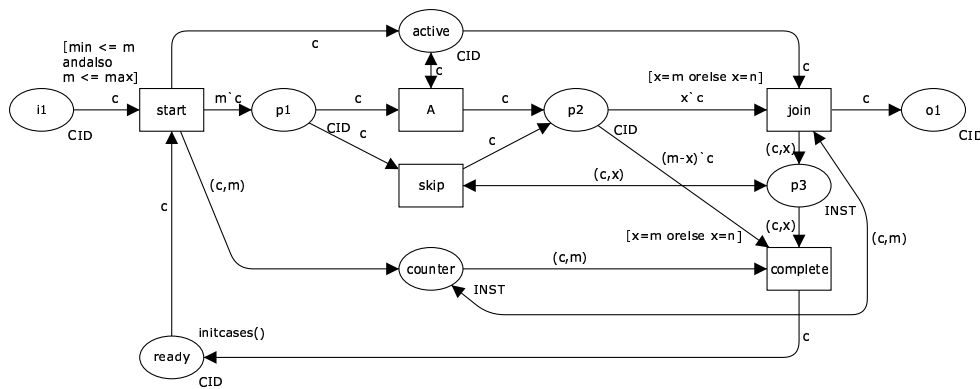


Figure 57: Cancelling partial join implementation for multiple instances

This pattern shares four context conditions with the *Static Partial Join for Multiple Instances* pattern: the number of concurrent activity instances (**m**) and the completion threshold (**n**) must be known before commencement, the number of instances that must complete for the join to be triggered (**n**) cannot be greater than the total

number of concurrent activity instances ( $m$ ), i.e.  $n \not\geq m$  and subsequent activities can be triggered as soon as the required completion threshold has been reached, however the final context condition is relaxed and the pattern is able to be re-enabled almost immediately after the completion threshold is reached as remaining activity instances are cancelled.

**Implementation** This pattern relies on the availability of a *Cancel Activity* or *Cancel Region* capability within an offering and at least one of these patterns needs to be supported for this pattern to be facilitated. As for WCP-34, both BPMN and XPDL appear to offer support for this pattern by associated an error type intermediate trigger with the multiple instance activity. Immediately following this activity is an activity that issues a cancel event effectively terminating any remaining activity instances once the first  $N$  of them have completed. However it is unclear how the *ComplexMLFlowCondition* should be specified to allow the cancellation to be triggered once  $N$  activity instances have completed.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern. An offering achieves partial support if there is any ambiguity associated with the implementation of the pattern.

### Pattern WCP-36 (Dynamic Partial Join for Multiple Instances)

**Description** Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. At any time, whilst instances are running, it is possible for additional instances to be initiated providing the ability to do so has not been disabled. A completion condition is specified which is evaluated each time an instance of the activity completes. Once the completion condition evaluates to true, the next activity in the process is triggered. Subsequent completions of the remaining activity instances are inconsequential and no new instances can be created.

#### Examples

- The despatch of an *oil rig* from factory to site involves numerous *transport shipment* activities. These occur concurrently and although sufficient activities are started to cover initial estimates of the required transport volumes, it is always possible for additional activities to be initiated if there is a shortfall in transportation requirements. Once 90% of the *transport shipment* activities are complete, the next activity (*invoice transport costs*) can commence. The remaining *transport shipment* activities continue until the whole rig has been transported.

**Motivation** This pattern is a variant of the *Multiple Instances without a priori Runtime Knowledge* pattern that allows the thread of execution to pass to subsequent activities once a specified completion condition is met. It allows the process to progress without requiring that all instances associated with a multiple instance activity have completed.

**Context** Figure 58 illustrates the operation of this pattern. The multiple instance activity is illustrated by transition A. At commencement, the number of instances

initially required is indicated by variable  $m$ . Additional instances may be added to this at any time via the **start instance** transition. At commencement, the pattern is in the active state. Once enough instances of activity **A** have completed and the join transition has fired, the next activity is enabled (illustrated via a token being placed in the output place **o1**) and the remaining instances of activity **A** run to completion before the **complete** transition is enabled. No new instances can be created at this time. Finally when all instances of **A** have completed, the pattern resets and can be re-enabled. An important feature of the pattern is the ability to disable further creation of activity instances at any time after the first instances has been created.

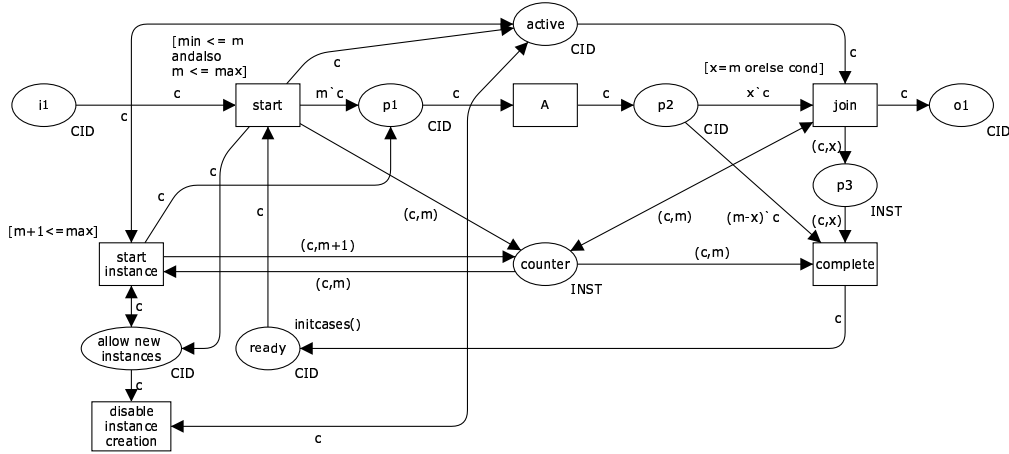


Figure 58: Dynamic partial join implementation for multiple instances

**Implementation** Of the offerings identified, only FLOWer provides support for the dynamic creation of multiple instance activities (via dynamic subplans), however it requires all of them to be completed before any completion conditions associated with a dynamic subplan (e.g. partial joins) can be evaluated and subsequent activities can be triggered. This is not considered to constitute support for this pattern.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. It achieves partial support if the creation of activity instances cannot be disabled once the first activity instance has commenced.

When defining the *Structured Synchronizing Merge* pattern (WCP-7) was introduced several context assumptions were made. Now we relax some of these assumptions.

### Pattern WCP-37 (Acyclic Synchronizing Merge)

**Description** The convergence of two or more branches which diverged earlier in the process into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is made on the basis of information locally available to the merge construct. This may be communicated directly to the

merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge.

**Example** Figure 59 provides an example of one solution to this pattern. It is based on the use of “true” and “false” tokens which are used to indicate whether a branch is enabled or not. After the divergence at transition A, one or both of the outgoing branches may be enabled. The determinant of whether the branch is enabled is that the token passed to the branch contains both the case id as well as a boolean variable which is “true” if the activities in the branch are to be executed, “false” otherwise. As the control-flow token is passed down a branch, if it is a “true” token, then each activity that receives the thread of control is executed otherwise it is skipped (illustrated by the execution of the bypass activity  $s1..sn$  associated with each activity). The *Synchronizing Merge*, which in this example is illustrated by transition E, can be evaluated when every incoming branch has delivered a token to the input places for the same case.

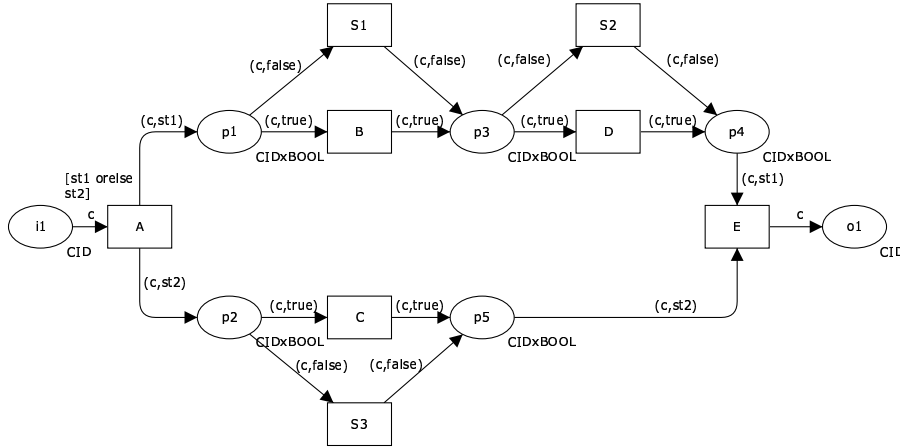


Figure 59: Acyclic synchronizing merge pattern

Another possible solution is provided by Rittgen [Rit99]. It involves the direct communication of the number of active branches from the preceding OR-Join divergence to the *Synchronizing Merge* so that it is able to determine when to fire.

**Motivation** The *Acyclic Synchronizing Merge* provides a deterministic semantics for the *Synchronizing Merge* pattern which does not rely on the process model being structured (as is required for the *Structured Synchronizing Merge*) but also does not require the use of non-local semantics in evaluating when the merge can fire.

**Context** The *Acyclic Synchronizing Merge* has two context conditions associated with its usage: (1) the merge construct must be able to determine how many incoming branches require synchronization based on local knowledge available to it during execution and (2) each active incoming branch must only contain at most one thread of control for a given process instance.

One of the main considerations which flows from this constraint is that it is not possible for this pattern to be used in loops (other than by including the entire subprocess containing both the preceding divergence(s) and the *Acyclic Synchronizing Merge* which is repeated).

**Implementation** WebSphere MQ, FLOWer, COSA, BPEL and EPCs provide support for this pattern. UML 2.0 ADs seems to provide support although there is some ambiguity over the actual JoinSpec configuration required.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern. If there is any ambiguity as to the manner in which the synchronization condition is specified, then it rates as partial support.

### Pattern WCP-38 (General Synchronizing Merge)

**Description** The convergence of two or more branches which diverged earlier in the process into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled or it is not possible that the branch will be enabled at any future time.

**Example** Figure 60 provides an example of the *General Synchronizing Merge* pattern. It shares a similar fundamental structure to the examples presented in Figures 7 and 59 for the other forms of OR-join however the feedback path from p4 to p1 involving F (which effectively embeds a “loop” within the process) means that it is not possible to model it either in a structured way or to use local information available to E to determine when the OR-join should be enabled.

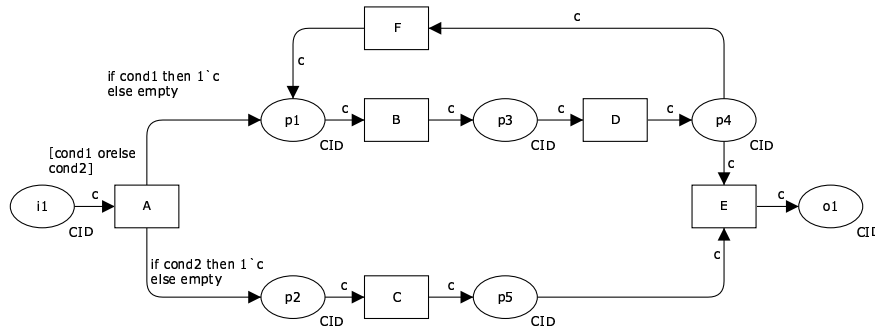


Figure 60: General synchronizing merge pattern

**Motivation** This pattern provides a general approach to the evaluation of the OR-join construct in workflow. It is able to be used in non-structured and highly concurrent workflow including process models that include looping structures.

**Context** This pattern provides general support for the OR-join construct that is widely utilised in modelling languages but is often only partially implemented or severely restricted in the form in which it can be used. The difficulty in implementing the *Synchronizing Merge* stems from the fact that its evaluation relies on non-local semantics [vdADK02] in order to determine when it can fire. In fact it is easy to see that this construct can lead to the “vicious circle paradox” [Kin06] where two OR-joins depend on one another.

The OR-join can only be enabled when the thread of control has been received from all incoming branches and it is certain that the remaining incoming branches which have not been enabled will never be enabled at any future time. Determination

of this fact requires a (computationally expensive) evaluation of possible future states for the current process instance.

**Implementation** FileNet is the only offering examined to support this pattern. An algorithm describing its implementation based on Reset-Nets is described in [WEvdAtH05] and has been used as the basis for the OR-join construct in the YAWL reference implementation [vdAtH05].

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that implements the context requirements for the pattern.

When discussing the *Interleaved Parallel Routing* pattern (WCP-17), we assumed the interleaved activities to be atomic. This can be generalized to critical sections where whole sets of activities should be executed on an atomic basis.

### Pattern WCP-39 (Critical Section)

**Description** Two or more connected subgraphs of a process model are identified as “critical sections”. At runtime for a given process instance, only activities in one of these “critical sections” can be active at any given time. Once execution of the activities in one “critical section” commences, it must complete before another “critical section” can commence.

#### Example

- Both the *take-deposit* and *final-payment* activities in the holiday booking process require the exclusive use of the *credit-card-processing* machine. Consequently only one of them can execute at any given time.

**Motivation** The *Critical Section* pattern provides a means of limiting two or more sections of a process from executing concurrently. Generally this is necessary if activities within this section require exclusive access to a common resource (either data or a physical resource) necessary for an activity to be completed. However, there are also regulatory situations (e.g. as part of due diligence or quality assurance processes) which necessitate that two activities do not occur simultaneously.

**Context** The operation of this pattern is illustrated in Figure 61. The mutex place serves to ensure that within a given process instance, only the sequence BD or CE can be active at any given time.

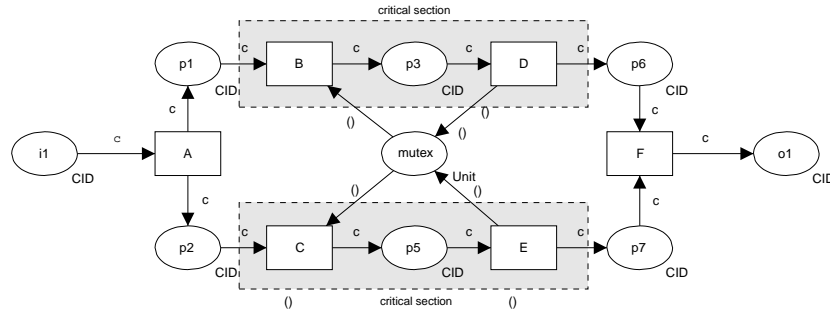


Figure 61: Critical section pattern



**Implementation** Although useful, this pattern is not widely supported amongst the offerings examined. BPEL allows it to be directly implemented through its serializable scope functionality. COSA supports this pattern by including a mutex place in the process model to prevent concurrent access to critical sections. FLOWer provides indirect support through the use of data elements as semaphores.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that implements the context requirements for the pattern. Where an offering is able to achieve similar functionality through configuration or extension of its existing constructs this qualifies as partial support.

### Pattern WCP-40 (Interleaved Routing)

**Description** Each member of a set of activities must be executed once. They can be executed in any order but no two activities can be executed at the same time (i.e. no two activities can be active for the same process instance at the same time). Once all of the activities have completed, the next activity in the process can be initiated.

#### Example

- The *check-oil*, *test-feeder*, *examine-main-unit* and *review-warranty* activities all need to be undertaken as part of the machine-service process. Only one of them can be undertaken at a time, however they can be executed in any order.

**Motivation** The *Interleaved Routing* pattern relaxes the partial ordering constraint that exists with the *Interleaved Parallel Routing* pattern and allows a sequence of activities to be executed in any order.

**Context** Figure 62 illustrates the operation of this pattern. After A is completed, activities B, C, D and E can be completed in any order. The **mutex** place ensures that only one of them can be executed at any time. After all of them have been completed, activity F can be undertaken.

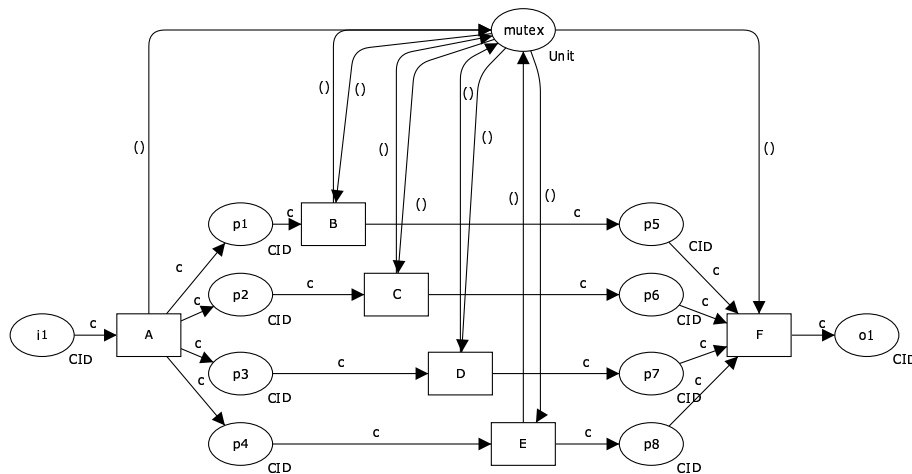


Figure 62: Interleaved routing pattern

There are two considerations associated with the use of this pattern: (1) for a given process instance, it is not possible for two activities from the set of activities

subject to interleaved routing to be executed at the same time and (2) activities must be initiated and completed on a sequential basis, in particular it is not possible to suspend one activity during its execution to work on another.

**Implementation** In order to effectively implement this pattern, an offering must have an integrated notion of state that is available during execution of the control-flow perspective. COSA has this from its Petri-Net foundation and is able to directly support the pattern. Other offerings lack this capability and hence are not able to directly support this pattern. BPEL (although not Oracle BPEL) can achieve similar effects using serializable scopes within the context of a <pick> construct. FLOWer has a distinct foundation to that inherent in other workflow products in which all activities in a case are always allocated to the same resource for completion hence interleaving of activity execution is guaranteed, however it is also possible for a resource to suspend an activity during execution to work on another hence the context conditions for this pattern are not fully satisfied. BPMN and XPDL indirectly support the pattern through the use of ad-hoc processes however it is unclear how it is possible to ensure that each activity in the ad-hoc sub-process is executed precisely once.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. An offering is rated as having partial support if it has limitations on the range of activities that can be coordinated (e.g. activities must be in the same process block) or if it cannot enforce that activities are executed precisely once or ensure activities are not able to be suspended once started whilst other activities in the interleave set are commenced.

The issue of synchronizing multiple branches within a process model has received a great deal of focus and is addressed by a number of patterns earlier in this paper. However the synchronization of multiple threads of execution within the same branch has not received the same degree of attention and consequently is the subject of the next two patterns.

### Pattern WCP-41 (Thread Merge)

**Description** At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution.

#### Example

- Instances of the *register-vehicle* activity run independently of each other and of other activities in the *Process Enquiry* process. They are created as needed. When ten of them have completed, the *process-registration-batch* activity should execute once to finalise the *vehicle registration system* records update.

**Motivation** This pattern provides a means of merging multiple threads within a given process instance. It is a counterpart to the *Thread Split* pattern which creates multiple execution threads along the same branch. In some situations, it can also be used in conjunction with the *Multiple Instances without Synchronization* pattern (WCP-12) however there is the requirement that each of the multiple instances execute along the same branch in the process.

**Context** The operation of this pattern is illustrated in Figure 63. A value for **numinsts** is included in the design-time process model and indicates the number of threads to be merged.

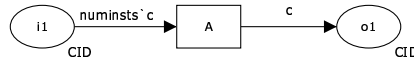


Figure 63: Thread merge pattern

There are two context considerations for this pattern: (1) the number of threads needing to be merged must be known at design-time and (2) only execution threads for the same process instance can be merged. If it is to be used to merge independent execution threads arising from some form of activity spawning (e.g. as a result of WCP-12), then it must be possible to identify the specific threads that need to be coalesced.

**Implementation** Implementation of this pattern implies that an offering is able to support the execution of processes in a non-safe context. This rules out the majority of the workflow systems examined from providing any tractable forms of implementation. BPMN and XPDL provide direct support for the pattern by including an activity after the spawned activity in which the **StartQuantity** attribute is set to the number of threads that need to be synchronized. The **StartQuantity** attribute specifies the number of incoming tokens required to start an activity. UML 2.0 ADs offer a similar behaviour via weights on **ActivityEdge** objects. BPEL provides an indirect means of implementation based on the correlation facility for feedback from the **<invoke>** action although some programmatic housekeeping is required to determine when synchronization should occur.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support for this pattern if it provides a construct that satisfies the context requirements. If any degree of programmatic extension is required to achieve the same behaviour, then the partial support rating applies.

### Pattern WCP-42 (Thread Split)

**Description** At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance.

#### Example

- At the completion of the *confirm paper receipt* activity, initiate three instances of the subsequent *independent peer review* activity.

**Motivation** This pattern provides a means of triggering multiple execution threads along a branch within a given process instance. It is a counterpart to the *Thread Merge* pattern which merges multiple execution threads along the same branch. Unless used in conjunction with the *Thread Merge* pattern, the execution threads will run independently to the end of the process.

**Context** The operation of this pattern is illustrated in Figure 64. A value for **numinsts** is included in the design-time process model and indicates the number of threads to be merged.

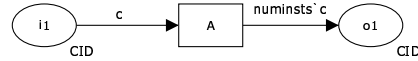


Figure 64: Thread merge pattern

There are two context considerations for this pattern: (1) the number of threads to be initiated must be known at design-time and (2) all threads must be initiated from the same point in the process model (i.e. they must flow along the same branch).

**Implementation** As with the *Thread Merge* pattern, implementation of this pattern implies that an offering is able to support the execution of processes in a non-safe context. This rules out the majority of the workflow systems examined from providing any tractable forms of implementation. BPMN and XPDL provide direct support for the pattern by allowing the Quantity of tokens flowing down the outgoing sequence flow from an activity at its conclusion to be specified. UML 2.0 ADs allow a similar behaviour to be achieved through the use of multiple outgoing edges from an activity to a MergeNode which then directs the various initiated threads of control down the same branch. BPEL indirectly allows the same effect to be achieved via the <invoke> action in conjunction with suitably specified correlation sets.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support for this pattern if it provides a construct that satisfies the context requirements. If any degree of programmatic extension is required to achieve the same behaviour, then the partial support rating applies.

Finally, we introduce the counterpart of the *Implicit Termination* pattern (WCP-11).

### Pattern WCP-43 (Explicit Termination)

**Description** A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instance is cancelled and the overall process instance is recorded as having completed successfully.

**Example** N/A.

**Motivation** The rationale for this pattern is that it represents an alternative means of defining when a process instance can be designated as complete. This is when the thread of control reaches a defined state within the process model. Typically this is denoted by a designated termination node at the end of the model.

**Context** There are two specific context conditions associated with this pattern: (1) every activity in a the process must be on a path from a defined starting node to a defined end node and (2) when the thread of control reaches the end node, the process is deemed to have completed successfully regardless of whether there are any activities in progress or remaining to be executed. For example, where a log is kept of process activity, the process instance would be recorded as completing successfully.

**Implementation** COSA, iPlanet, SAP Workflow, BPMN, XPDL and UML 2.0 ADs support this pattern although other than iPlanet, none of these offerings enforce that there is a single end node.

**Issues** One consideration that does arise where a process model has multiple end nodes is whether it can be transformed to one with a single end node.

**Solutions** For simple process models, it may be possible to simply replace all of the end nodes for a process with links to an OR-join which then links to a single final node. However, it is less clear for more complex process models involving multiple instance activities whether they are always able to be converted to a model with a single terminating node. Potential solutions to this are discussed at length in [KtHvdA03].

**Evaluation Criteria** An offering achieves full support for this pattern if it demonstrates that it can meet the context requirements for the pattern.

This concludes the discussion of new patterns relevant to the control-flow perspective of process-aware information systems. We now move on to a comprehensive discussion of the degree of support for each of these patterns in individual commercial offerings.

## 5 Evaluating Process-Aware Information Systems

In this section, we consider the evaluation results obtained from a detailed analysis of the control-flow patterns across fourteen commercial offerings. The products examined include workflow systems, a case handling system, business process execution languages and business process modelling formalisms. The specific products/languages examined were:

- Staffware Process Suite 10;
- IBM WebSphere MQ Workflow 3.4;
- FLOWer 3.5.1;
- COSA 5.1;
- Sun ONE iPlanet Integration Server 3.0;
- SAP Workflow version 4.6c
- FileNet P8 BPM Suite version 3.5
- BPEL version 1.1;
- WebSphere Integration Developer 6.0.2, the development environment for the Business Process Choreographer (BPC) part of WebSphere Process Server v6.0.2;
- Oracle BPEL v10.1.2;
- BPMN version 1.0;
- XPDL version 2.0;
- UML 2.0 Activity Diagrams; and
- EPCs as implemented by ARIS Toolset 6.2.

Although many of the evaluation results reinforce observations regarding pattern implementation that have been made over the past seven years, it is interesting to observe some of the trends that have become evident in recent product offerings. These are brought into sharper relief through the augmented set of control-flow patterns described in this paper.

Traditionally workflow systems have employed proprietary approaches both to the range of concepts that can be expressed in design-time process models and also to the way that those models are enacted at run-time. The implementation of concepts such as threads of control, join semantics and loops differ markedly between offerings. The fundamental process model employed by specific products has been a particularly vague area. This seems to have changed in recent offerings towards a more formally defined and better understood model with BPEL, BPMN, XPDL and UML 2.0 ADs all claiming an underlying execution model that is “token-based”.

Although the revised definitions of the original patterns includes some more restrictive definitions – particularly for patterns such as *Structured Synchronizing Merge*, *Structured Discriminator*, *Multiple Instances with Design-Time Knowledge* and *Interleaved Parallel Routing* – these patterns continue to be relatively widely supported. In particular all of the basic patterns (WCP-1 to WCP-5) are still supported by all offerings examined.

The revised definition of the the *Structured Synchronizing Merge* tends to favour block structured languages such as WebSphere MQ and BPEL although it is also supported by COSA, FileNet, BPMN, XPDL and EPCs despite the restrictions that it implies on the manner in which it can be used. Although more flexible variants of it can be delineated in the form of the *Acyclic Synchronizing Merge* and the *General*

Table 1: Product Evaluations – Original Patterns (WCP1 - WCP20)

Pattern	Staffware	WebSphere MQ	FLOWer	COSA	iPlanet	SAP Workflow	FileNet	BPEL	WebSphere BPEL	Oracle BPEL	BPMN	XPDL	UML ADs	EPC
1 (seq)	+	+	+	+	+	+	+	+	+	+	+	+	+	+
2 (par-spl)	+	+	+	+	+	+	+	+	+	+	+	+	+	+
3 (synch)	+	+	+	+	+	+	+	+	+	+	+	+	+	+
4 (ex-ch)	+	+	+	+	+	+	+	+	+	+	+	+	+	+
5 (simple-m)	+	+	+	+	+	+	+	+	+	+	+	+	+	+
6 (m-choice)	-	+	+	+	+	-	+	+	+	+	+	+	+	+
7 (s-sync-m)	-	+	+	-	-	-	+	+	+	+	+	+	-	+
8 (multi-m)	-	-	+/-	+/-	+	-	+	-	-	-	+	+	+	-
9 (s-disc)	-	-	-	-	+	+/-	-	-	-	-	+/-	+/-	+/-	-
10 (arb-c)	+	-	-	+	+	-	+	-	-	-	+	+	+	+
11 (impl-t)	+	+	+	-	-	-	+	+	+	+	+	+	+	+
12 (mi-no-s)	+	-	+	+	+	+/-	+	+	+	+	+	+	+	-
13 (mi-dt)	+	-	+	-	-	+	-	-	-	+	+	+	+	-
14 (mi-rt)	+	-	+	-	-	+	-	-	-	+	+	+	+	-
15 (mi-no)	-	-	+	-	-	-	-	-	-	-	-	-	-	-
16 (def-c)	-	-	+	+	-	-	+/-	+	+	+	+	+	+	-
17 (int-par)	-	-	+/-	+	-	-	-	+/-	+/-	-	-	-	-	-
18 (milest)	-	-	+/-	+	-	-	-	-	-	-	-	-	-	-
19 (can-a)	+	-	+/-	+	+	+	+	+	+	+	+	+	+	-
20 (can-c)	-	-	+/-	-	-	+	+	+	+	+	+	+	+	-

*Synchronizing Merge*, these forms are only minimally supported. Similar observations can be drawn for the *Structured Discriminator* and the *Structured Partial Join* which have minimal support particularly amongst offerings with an actual execution environment and the other flexible forms of these patterns (i.e. *Blocking* and *Cancelling Discriminator* and *Blocking* and *Cancelling Partial Join*) which have extremely minimal support overall.

An interesting observation arising from these patterns is that despite the claims in regard to the existence of a deterministic mapping from business modelling languages such as BPMN and XPDL to the execution language BPEL, there are a number of patterns such as the *Multi-Merge*, all forms of the *Discriminator* and the *Partial Join* and *Arbitrary Cycles* which are supported in the former languages but not in BPEL begging the question of how these patterns can actually be implemented. This finding is consistent with other research [OvdADtH06] and reflects the fact that *modelling languages tend to fare better in patterns evaluations than actual operational products*

*because they are not faced with the burden of actually having to implement everything that they are able to model!*

In general, the evaluation results for the two BPEL offerings - WebSphere Integration Developer and Oracle BPEL – indicate that they provide relatively faithful implementations of the BPEL specification. One noteworthy exception to this is the additional <flowN> construct provided by Oracle BPEL which allows it to implement several of the multiple instance patterns which other BPEL variants are not able to support.

The form of the *Arbitrary Cycles* pattern tends to mitigate against it being supported in block structured languages such as WebSphere MQ or BPEL. However the more restricted form of repetition – the *Structured Loop* pattern – is quite widely supported in a number of offerings including WebSphere MQ, FLOWer, iPlanet, FileNet, BPEL, BPMN and XPDL.

Other than for the *Multiple Instances without Synchronization* pattern which is widely supported, there continues to be minimal support for managing controlled activity concurrency in its various forms (e.g. *Multiple Instances with Design-Time/with Run-time/without Run-Time Knowledge*, various forms of *Partial Join for Multiple Instances*, *Cancel Multiple Instance Activity*). Staffware, FLOWer and Oracle BPEL are the only offerings that provide some degree of runtime support for the various multiple instance patterns whilst BPMN and XPDL provide the ability to capture them in design-time models.

The *Explicit Termination* pattern has been introduced to describe situations where there is a dedicated termination node in a process rather than the assumption that a process instance terminates when there is no more work remaining (i.e. *Implicit Termination*). Most offerings support one or the other of these patterns, but interestingly BPMN, XPDL and UML 2.0 ADs support *both* patterns.

In terms of state-based patterns, effective implementation of the *Deferred Choice* favours those offerings with a deterministic underpinning (e.g. token-based) such as COSA, BPEL, BPMN, XPDL and UML 2.0 ADs. Only COSA directly supports *Interleaved Parallel Routing* and in general, it seems that an integrated notion of state is required to effectively implement this pattern. Similar comments apply to the *Milestone* pattern. Where the partial ordering requirement is relaxed allowing for arbitrary execution order (i.e. *Interleaved Routing*), BPEL (although not Oracle BPEL) is also able to provide support. The need to ensure that activities are not running in parallel is not present in FLOWer. However, the lack of true concurrency (other than interleaving) rules out full support for this pattern. Interestingly, the ad-hoc activity construct in BPMN and XPDL provides a means of indirectly achieving this pattern but does not faithfully ensure that each activity is run precisely once. The *Critical Section* pattern is only supported by COSA and BPEL.

Cancellation patterns are another area worthy of comment. *Cancel Activity* is widely supported although WebSphere MQ is a notable exception. Of interest is the distinction between withdrawing an activity prior to it commencing (supported by Staffware, COSA, SAP) and cancelling it during execution (supported by BPEL, BPMN, XPDL and UML 2.0 ADs). Similarly the *Cancel Case* pattern is widely supported (by SAP, FileNet, BPEL, BPMN, XPDL, UML 2.0 ADs), but cancellation of an arbitrary region (i.e. *Cancel Region*) is not, only fully supported by UML 2.0 ADs.



Table 2: Product Evaluations – New Patterns (WCP21 - WCP43)

Pattern	Staffware	WebSphere	FLOWer	COSA	iPlanet	SAP Workflow	FileNet	BPEL	WebSphere BPEL	Oracle BPEL	BPMN	XPDL	UML ADs	EPC
21 (str-l)	-	+	+	-	+	+	+	+	+	+	+	+	+	-
22 (recur)	+	+	-	+	+	+	-	-	-	-	-	-	-	-
23 (t-trig)	+	-	-	+	-	+	-	-	-	-	-	-	+	-
24 (p-trig)	-	-	+	+	-	+	+	+	+	+	+	+	+	+/-
25 (can-r)	-	-	-	+/-	-	-	-	+/-	+/-	+/-	+/-	+/-	+	-
26 (can-mi)	+	-	-	-	-	+	-	-	-	+	+	+	+	-
27 (comp-mi)	-	-	+/-	-	-	-	-	-	-	-	-	-	-	-
28 (b-disc)	-	-	-	-	-	-	-	-	-	-	+/-	+/-	+/-	-
29 (c-disc)	-	-	-	-	-	+	-	-	-	-	+	+	+	-
30 (s-pjoin)	-	-	-	-	+	+/-	-	-	-	-	+/-	+/-	+/-	-
31 (b-pjoin)	-	-	-	-	-	-	-	-	-	-	+/-	+/-	+/-	-
32 (c-pjoin)	-	-	-	-	-	+	-	-	-	-	+/-	+/-	+	-
33 (g-and-join)	-	-	-	-	-	-	+	-	-	-	+	+	-	+/-
34 (st-pjoin-mi)	-	-	-	-	-	-	-	-	-	-	+/-	+/-	-	-
35 (c-pjoin-mi)	-	-	-	-	-	-	-	-	-	-	+/-	+/-	-	-
36 (dyn-pjoin-mi)	-	-	-	-	-	-	-	-	-	-	-	-	-	-
37 (a-sync-m)	-	+	+	+	-	-	-	+	+	+	-	-	+/-	+
38 (g-sync-m)	-	-	-	-	-	-	+	-	-	-	-	-	-	-
39 (crit-sec)	-	-	+/-	+	-	-	-	+	+	+	-	-	-	-
40 (int-rout)	-	-	+/-	+	-	-	-	+	+	-	+/-	+/-	-	-
41 (tm)	-	-	-	-	-	-	-	+/-	+/-	+/-	+	+	+	-
42 (ts)	-	-	-	-	-	-	-	+/-	+/-	+/-	+	+	+	-
43 (exp-t)	-	-	-	+	+	+	-	-	-	-	+	+	+	-

Trigger-based patterns which offer the ability for external factors to control process execution are also widely supported. Staffware implements transient triggers whilst persistent triggers are offered by FileNet, BPEL, BPMN and XPDL. COSA, SAP Workflow and UML 2.0 ADs implement both forms of trigger.

## 6 Relationships Between Patterns

In his seminal work *A Patterns Language* [AIS77], Alexander advanced the proposition that one of the most interesting aspects of a patterns taxonomy in a given domain lays in the relationships that exist between the individual patterns. For the control-flow patterns, there are two relationships that of of interest: the *specialization* relationship which identifies where one pattern is a more restricted form of another pattern and the *composition* relationship which identifies groups of patterns which can be used together to provide the same functionality as another pattern. These are discussed in further details below.

### 6.1 Specialization Relationships

A specialization relationship exists between two patterns where one is a more restricted form of another i.e. it supports a subset of the behaviours of another or operates in a more restricted situation. As an example of this, the *Multi-choice* (WCP-6) is a more general form of the *Parallel Split* (WCP-2) pattern. Indeed, the *Multi-choice* can be substituted for the *Parallel Split* without any resultant loss of functionality. For some patterns such as the *Synchronization* (WCP-2) or several of the *Discriminator* variants, there are several possible specializations. Table 3 identifies each of these relationships.

Table 3: Specialization relationships between workflow patterns

Nr	Specialized Pattern Name	General Pattern
3	Synchronization	31. Blocking Partial Join
		32. Cancelling Partial Join (only where N=M)
		33. Generalized AND-Join
		38. General Synchronizing Merge
4	Exclusive Choice	6. Multi-Choice
5	Simple Merge	8. Multi-Merge
6	Multi-choice	2. Parallel Split
7	Structured Synchronizing Merge	37. Acyclic Synchronizing Merge
9	Structured Discriminator	28. Blocking Discriminator
		30. Structured Partial Join
13	MIIs with a priori D/T Knowledge	14. MIIs with a priori R/T Knowledge
14	MIIs with a priori R/T Knowledge	15. MIIs without a priori R/T Knowledge
19	Cancel Activity	25. Cancel Region
20	Cancel Case	25. Cancel Region
26	Cancel Multiple Instance Activity	19. Cancel Activity
27	Complete MI Activity	26. Cancel MI Activity
28	Blocking Discriminator	31. Blocking Partial Join
29	Cancelling Discriminator	32. Cancelling Partial Join
30	Structured Partial Join	31. Blocking Partial Join
34	Static Partial Join for MIIs	36. Dynamic Partial Join for MIIs
37	Acyclic Synchronizing Merge	38. General Synchronizing Merge
40	Interleaved Routing	17. Interleaved Parallel Routing

The specialization relationships forms a hierarchy that can also be illustrated graphically as shown in Figure 6.2. Note that implied transitive relationships have not been included for the sake of clarity.

## 6.2 Composition Relationships

In some cases, it is possible that a pattern can be implemented through a combination of two (or more) other patterns. For example, the *Cancelling Discriminator* (WCP-29) can be implemented using a combination of the *Cancel Region* (WCP-25) and *Blocking Discriminator* (WCP-28) patterns. This type of relationship is termed a *composition* relationship. Table 4 identifies the most significant examples of these relationships between patterns although the list is not exhaustive as differing implementation approaches for individual patterns may result alternate composition relationships.

Table 4: Composition relationships between workflow patterns

Nr	Pattern Name	Composed From
6	Multi-Choice	2. Parallel Split in conjunction with 4. Exclusive Choice
10	Arbitrary Cycles <sup>9</sup>	4. Exclusive Choice together with 5. Simple Merge
12	MI without Synchronization <sup>4</sup>	6. Multi-Choice together with 8. Multi-Merge
21	Structured Loop <sup>4</sup>	4. Exclusive Choice together with 8. Multi-Merge
29	Cancelling Discriminator	25. Cancel Region together with 28. Blocking Discriminator
32	Cancelling Partial Join	25. Cancel Region together with 31. Blocking Partial Join
35	Cancelling Partial Join for MIs	34. Static Partial Join for MIs together with 26. Cancel MI Activity

To provide some context to the complete set of relationships, the diagram in Figure 6.2 illustrates all of the relationships identified in a single diagram. Specializations are shown with a solid line and composition relationships with a dashed line. Through this diagram it is possible to get a clearer idea of the relationships between various patterns and it is an interesting observation that this connected diagram includes 32 of the 43 patterns identified.



## 7 Conclusions

In this paper, we have reviewed the original twenty control-flow patterns and have provided a precise definition of their operation and the context in which they are intended to be utilized. We have also reviewed their implementation in fourteen commercial offerings. The original set of patterns remains as valid as they were when first identified seven years ago and all of them have been retained as a result of this survey. One of the major contributions of this research effort is the establishment of a clear set of operational semantics for each of pattern. This is intended to remove any potential for ambiguity or misinterpretation that may have existed with earlier documentation.

As a consequence of refining the definition of each pattern, it has been possible to identify twenty three additional patterns relevant to the control-flow perspective. Some of these are a result of having achieved a better understanding of the original patterns and recognizing that in some cases individual patterns could potentially have more than one interpretation, whilst other patterns reflect our deeper understanding of the control-flow perspective and recognize some gaps that may have existed in the original set.

An important observation that we draw from this research is the disparity that exists between modelling languages and actual product implementations in terms of the number of patterns that they support. Whilst several of the contemporary modelling formalisms (BPMN, XPDL, UML 2.0 ADs) are able to capture a broad range of patterns, it is interesting to note that they do not demonstrate how these patterns will actually be realized in practice. This opens up an inherent contradiction where a particular offering claims to be compliant with a particular modelling formalism but supports *less* patterns. Similar difficulties exist with proposed mappings between these modelling languages and particular execution tools as they cannot claim to be complete.

Patterns identification in any domain is by definition an experiential activity and it is not possible to guarantee the completeness of any patterns collection. However, it is exactly this approach to knowledge gathering that ensures that patterns remain relevant to the domain that they seek to describe. By pairing this strategy with a formal approach to describing the concepts that are discovered and continually refining these definitions on a long-term basis, the *Workflow Patterns Initiative* aims to provide the most comprehensive description available of concepts relevant to the modelling and execution of workflow systems both now and on an ongoing basis.

In tandem with this research, the *YAWL System* ([www.yawl-system.com](http://www.yawl-system.com)) provides an open-source reference model for workflow systems based on the workflow patterns. YAWL currently supports 32 of the control-flow patterns. It is subject to ongoing development which continues to broaden the range of patterns that it implements (not only from the control-flow perspective, but also for the data, resource and exception perspectives as well) and also to provide a vehicle which demonstrates how specific patterns might be realized and utilized in practice.

**Acknowledgements** We would like to thank David Edmond, Marlon Dumas, Petia Wohed, Boudewijn van Dongen and the members of the BPM Group at QUT and the BETA Research Centre at TU/e for their useful comments that have helped to improve this paper. We also thank the vendors and consultants that assisted in reviewing the results and provided some valuable feedback.

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

## References

- [AIS77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.
- [BDtH05] A. Barros, M. Dumas, and A.H.M. ter Hofstede. Service interaction patterns. In W.M P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the 3rd International Conference on Business Process Management (BPM'2005)*, pages 302–318, Nancy, France, 2005. Springer Verlag.
- [BMR<sup>+</sup>96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns Volume 1*. Wiley, Chichester, 1996.
- [DtH01] M. Dumas and A.H.M. ter Hofstede. UML activity diagrams as a workflow specification language. In M. Gogolla and C. Kobryn, editors, *Proceedings of the Fourth International Conference on the Unified Modeling Language (UML 2001)*, volume 2185 of *Lecture Notes in Computer Science*, pages 76–90, Toronto, Canada, 2001. Springer.
- [EP00] H.E. Eriksson and M. Penker. *Business Modeling with UML*. OMG Press, New York, 2000.
- [Fow96] M. Fowler. *Analysis Patterns : Reusable Object Models*. Addison-Wesley, Boston, 1996.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, USA, 1995.
- [Hay95] D.C. Hay. *Data Model Patterns, Conventions of Thought*. Dorset House, New York, 1995.
- [HW04] G. Hohpe and B. Woolf. *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, 2004.
- [JB96] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. Thomson Computer Press, London, UK, 1996.
- [Jen97] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
- [Kie03] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, 2003.
- [Kin06] E. Kindler. On the semantics of EPCs: Resolving the vicious circle. *Data and Knowledge Engineering*, 56(1):23–40, 2006.



- [KtHB00] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In B. Wangler and L. Bergman, editors, *Proceedings of the 12th International Conference on Advanced Information Systems Engineering CAiSE 2000*, volume 1789 of *Lecture Notes in Computer Science*, Stockholm, Sweden, 2000. Springer.
- [KtHvdA03] B. Kiepuszewski, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003.
- [MvdAtHR06] N. Mulyar, W.M.P. van der Aalst, A.H.M. ter Hofstede, and N. Russell. A critical analysis of the 20 classical workflow control-flow patterns. Technical report, BPM Center Report BPM-06-18, 2006.
- [OvdADtH06] C. Ouyang, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Translating BPMN to BPEL. Technical report, BPM Center Report BPM-06-02, 2006.
- [Rit99] P. Rittgen. Object-oriented analysis using event-driven method chains. In *Proceedings of the MobIS-Fachtagung 1999*, volume 6 of *Rundbrief der GI-Fachgruppe 5.10, (in German)*, pages 8–23, Bamberg, Germany, 1999.
- [RtHEvdA05] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005)*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368, Klagenfurt, Austria, 2005. Springer.
- [RvdAtH06] N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Workflow exception patterns. In E. Dubois and K. Pohl, editors, *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE’06)*, volume 4001 of *Lecture Notes in Computer Science*, pages 288–302, Luxembourg, Luxembourg, 2006. Springer.
- [RvdAtHE05] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow resource patterns: Identification, representation and tool support. In O. Pastor and J. Falcao é Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE’05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232, Porto, Portugal, 2005. Springer.
- [RvdAtHW06] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and P. Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In M. Stumptner, S. Hartmann, and Y. Kiyoki, editors, *Proceedings of the Third Asia-Pacific Conference on Conceptual Modelling (APCCM2006)*, volume 53 of *CRPIT*, pages 95–104, Hobart, Australia, 2006. ACS.

- [SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern Oriented-Software Architecture: Patterns for Concurrent and Networked Objects Volume 2*. Wiley, Chichester, 2000.
- [vdA01] W.M.P. van der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Systems Frontiers*, 3(3):297–317, 2001.
- [vdABtHK00] W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, and B. Kiepuszewski. Advanced workflow patterns. In O. Etzion and P. Scheuermann, editors, *Proceedings of the Fifth IFCIS International Conference on Cooperative Information Systems (CoopIS'2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29, Eilat, Israel, 2000. Springer.
- [vdADK02] W.M.P. van der Aalst, J. Desel, and E. Kindler. On the semantics of EPCs: A vicious circle. In M. Rump and F.J. Nuttgens, editors, *Proceedings of the EPK 2002: Business Process Management using EPCs*, pages 71–80, Trier, Germany, 2002. Gesellschaft für Informatik.
- [vdAtH05] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
- [vdAtHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.
- [WEvdAtH05] 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, *Proceedings of the 26th International Conference on Application and Theory of Petri nets and Other Models of Concurrency (Petri Nets 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 423–443, Miami, USA, 2005. Springer-Verlag.
- [Whi04] 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.
- [Wor99] Workflow Management Coalition. Terminology and glossary. Technical Report Document Number WFMC-TC-1011, Issue 3.0, 1999. [http://www.wfmc.org/standards/docs/TC-1011-term-glossary\\_v3.pdf](http://www.wfmc.org/standards/docs/TC-1011-term-glossary_v3.pdf).
- [WPDtH03] 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 O. Camp and M. Piattini, editors, *Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS 2003)*, volume 3, pages 174–184, Angers, France, 2003. Escola Superior de Tecnologia do Instituto Politecnico de Setubal.

- [WvdAD<sup>+</sup>05a] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and N. Russell. Pattern-based analysis of BPMN - an extensive evaluation of the control-flow, the data and the resource perspectives. Technical Report BPM Center Report BPM-05-26, BPMcenter.org, 2005.
- [WvdAD<sup>+</sup>05b] 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. In L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *Proceedings of the 25th International Conference on Conceptual Modeling (ER'2005)*, volume 3716 of *Lecture Notes in Computer Science*, pages 63–78, Klagenfurt, Austria, 2005. Springer.
- [WvdADtH03] 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 I.Y. Song, S.W. Liddle, T.W. Ling, and P. Scheuermann, editors, *Proceedings of the 22nd International Conference on Conceptual Modeling (ER'2003)*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215, Chicago, IL, USA, 2003. Springer.

## A Staffware

This evaluation is based on Staffware Process Suite 10.

Pattern	Score	Motivation
1 (seq)	+	Directly supported by arcs (drawn as lines from left to right) connecting steps.
2 (par-spl)	+	Supported through a step construct that has multiple outgoing arcs.
3 (synch)	+	Supported through the wait step construct. Note that the wait step has one preceding step (solid line) and possibly multiple steps it is waiting for (dashed lines). The difference between the preceding step and the steps it is waiting for becomes only visible in a loop situation. In a loop the wait step only waits for the preceding step (solid line) and no longer has to wait for the other steps. The only way to get a “normal” synchronization is a loop is to explicitly reset the step in a loop using the “SETSETSTATUS” function.
4 (ex-ch)	+	Supported through the condition construct modeling a binary decision. The decision is evaluated using a Boolean function and has only one or two outgoing arcs.
5 (simple-m)	+	Supported through a step construct that has multiple input arcs. Also a router can be used to model such a join. Note that Staffware only merges flows that are safe, i.e., if multiple triggers arrive at a step, only one trigger is retained.
6 (m-choice)	–	Not supported. The condition construct can only model a binary decision.
7 (s-sync-m)	–	Not supported. The wait step synchronizes flows and all other steps get enabled after receiving the first trigger.
8 (multi-m)	–	Not supported. It is not possible to trigger a step twice. Where this occurs, the second thread cancels the first.
9 (s-disc)	–	Not supported (see above).
10 (arb-c)	+	In general, unstructured loops are supported although there are some syntactical limitations.
11 (impl-t)	+	Directly supported. A workflow case terminates if all of its branches have terminated. A stop symbol can be used to indicate the end of each branch.

Pattern	Score	Motivation
12 (mi-no-s)	+	Staffware supports static and dynamic subprocedure steps. The static subprocedure step is simply a step corresponding to a subprocess. When Staffware processes a dynamic sub-procedure step, it looks at the array field that has been defined for the sub-procedures to start. This array field may contain no data (i.e. no sub-procedures need to be started) or multiple data elements (i.e. multiple sub-procedures need to be started) concurrently.
13 (mi-dt)	+	Supported using the dynamic subprocedure step.
14 (mi-rt)	+	Supported using the dynamic subprocedure step.
15 (mi-no)	–	Not supported. The number of instances is based on the array values at the moment the step is executed and cannot be changed later.
16 (def-c)	–	Not supported. No state support in the process model. Although there is a workaround based on a parallel split and withdraw actions, it is not safe.
17 (int-par)	–	Not supported. There is no way to interleave steps without specifying an order.
18 (milest)	–	Not supported. There is no notion of state.
19 (can-a)	+	Supported through the withdraw construct, i.e., a line entering a step from above.
20 (can-c)	–	Not directly supported, steps can be called via API calls.
21 (str-l)	–	Not supported. Loops can only be created in the graphical editor.
22 (recur)	+	Using the dynamic subprocedure step it is possible to call any procedure. However, it is unclear whether this is inadvertant rather than intended behavior (i.e. a backdoor).
23 (t-trig)	+	The event step construct allows external signals to trigger steps, cases and also to resume suspended steps.
24 (p-trig)	–	Not supported. However, by adding a dummy step, a transient trigger can be made persistent.
25 (can-r)	–	Not supported. Although steps can be withdrawn (providing they have not already commenced), it is not possible to specify a region, i.e., a withdraw for each individual step is required and complications may occur in regard to routing elements (e.g., wait steps).
26 (can-mi)	+	It is possible to withdraw subprocedure steps. However, in this case the sub-procedure is terminated prematurely without transferring any data back.
27 (comp-mi)	–	Not supported.

Pattern	Score	Motivation
28 (b-disc)	–	Not supported. As indicated above only the XOR-join and AND-join are possible using normal steps and wait steps respectively.
29 (c-disc)	–	Not supported (see above).
30 (s-pjoin)	–	Not supported.
31 (b-pjoin)	–	Not supported.
32 (c-pjoin)	–	Although simple versions of this pattern (e.g. 1-out-of-2 join) can constructed using withdrawn actions, the solution is not safe and does not scale up well to more complex joins.
33 (g-and-join)	–	Not supported. Although join constructs require triggering on all branches, subsequent triggers received on branches before the join has completed are lost.
34 (st-pjoin-mi)	–	Not supported. The dynamic subprocedure is specified using an array. It is only stopped after a failure or a withdraw of the complete subprocedure. There is no way to pass on control earlier.
35 (c-pjoin-mi)	–	Not supported.
36 (dyn-pjoin-mi)	–	Not supported.
37 (a-sync-m)	–	Not supported. The concept of a step waiting for all preceding activities to finish when they are optional is not possible in any form.
38 (g-sync-m)	–	Not supported.
39 (crit-sec)	–	Not supported. Since Staffware always immediately schedules subsequent activities, there is no way of temporarily blocking them. Note that withdrawing a step does not solve the problem.
40 (int-rout)	–	Not supported. The only way to model this is to enumerate sequences and explicitly select paths through conditions.
41 (tm)	–	No support for user-specified thread merging. The system automatically merges distinct control threads which reach the same step in a process instance.
42 (ts)	–	No support for user-specified thread merging. The system automatically merges distinct control threads which reach the same step in a process instance.
43 (exp-t)	–	Not supported. A workflow case terminates when all of its branches have terminated.

## B WebSphere MQ

This evaluation is based on WebSphere MQ Workflow 3.4.

Pattern	Score	Motivation
1 (seq)	+	Directly supported by arcs connecting process, program and block activities.
2 (par-spl)	+	Supported through multiple outgoing arcs from an activity.
3 (synch)	+	Supported by specifying start conditions on an activity.
4 (ex-ch)	+	Supported through the use of exclusive conditions on transitions.
5 (simple-m)	+	Supported by specifying start conditions on an activity.
6 (m-choice)	+	Supported through the use of (non-exclusive) conditions on transitions.
7 (s-sync-m)	+	Supported by specifying start conditions on an activity.
8 (multi-m)	–	Not supported. An activity can only be triggered once, either when one or all of the incoming connectors evaluate to true.
9 (s-disc)	–	Not supported. The evaluation of start conditions for an activity only occurs when all preceding activities have completed.
10 (arb-c)	–	Not supported. Process models are block-structured.
11 (impl-t)	+	Directly supported.
12 (mi-no-s)	–	Although it is possible to replicate an activity by including it in a block activity with an exit condition that is satisfied when all instances have completed, it is not possible for these instances to run concurrently.
13 (mi-dt)	–	Not supported. No construct for designating multiple instances of an activity in the design-time model.
14 (mi-rt)	–	Not supported. No means of facilitating multiple instances of an activity at runtime.
15 (mi-no)	–	Not supported. No means of facilitating multiple instances of an activity at runtime.
16 (def-c)	–	Not supported. There is no means of selecting that one out of a set of possible activities be executed (and the other activities be withdrawn).
17 (int-par)	–	Not supported. There is no way to interleave activities without specifying an order.
18 (milest)	–	Not supported. There is no inherent notion of state.
19 (can-a)	–	Not supported. There is no means of denoting activity cancellation with a process model.
20 (can-c)	–	Not supported. There is no means of cancelling an entire process instance.

Pattern	Score	Motivation
21 (str-l)	+	Post-tested loops are supported by the block construct.
22 (recur)	+	Directly supported. Recursive definition of process and block activities is possible.
23 (t-trig)	–	Not supported. There is no means of triggering an activity from outside the process instance.
24 (p-trig)	–	Not supported. There is no means of triggering an activity from outside the process instance.
25 (can-r)	–	Not supported. A set of activities cannot be cancelled.
26 (can-mi)	–	Not supported. There is no direct support for multiple instance activities.
27 (comp-mi)	–	Not supported. There is no direct support for multiple instance activities.
28 (b-disc)	–	Not supported. The evaluation of start conditions for an activity only occurs when all preceding activities have completed.
29 (c-disc)	–	Not supported. There is no support for the discriminator pattern or any ability to cancel a set of (preceding) activities.
30 (s-pjoin)	–	Not supported. There is no direct support for multiple instance activities.
31 (b-pjoin)	–	Not supported. There is no direct support for multiple instance activities.
32 (c-pjoin)	–	Not supported. There is no direct support for multiple instance activities.
33 (g-and-join)	–	Not supported. Process models are inherently block structured and an activity cannot receive multiple threads of control from the same incoming branch.
34 (st-pjoin-mi)	–	Not supported. There is no direct support for multiple instance activities.
35 (c-pjoin-mi)	–	Not supported. There is no direct support for multiple instance activities.
36 (dyn-pjoin-mi)	–	Not supported. There is no direct support for multiple instance activities.
37 (a-sync-m)	+	Supported through the use of dead path elimination where “true” and “false” tokens are passed down branches that are and are not enabled respectively. This allow the OR-join to determine when it should fire.
38 (g-sync-m)	–	Not supported. No ability to determine when an OR-join should fire based on an overall assessment of the state of a process instance.
39 (crit-sec)	–	Not supported. Subsequent activities are scheduled immediately thus removing any potential for restricting concurrent execution of activities.



<b>Pattern</b>	<b>Score</b>	<b>Motivation</b>
40 (int-rout)	–	Not supported. There is no way to interleave activities without actually enumerating all possible execution sequences within the process model and selecting one of them at runtime.
41 (tm)	–	No support. Process models are block structured and safe.
42 (ts)	–	No support. Process models are block structured and safe.
43 (exp-t)	–	Not supported. Process instances terminate when there is no remaining work.

## C FLOWer

This evaluation is based on FLOWer 3.5.1.

Pattern	Score	Motivation
1 (seq)	+	Supported through arcs connecting plan elements.
2 (par-spl)	+	Nodes in static, dynamic and sequential subplans have an AND-split semantics.
3 (synch)	+	Nodes in static, dynamic and sequential subplans have an AND-join semantics.
4 (ex-ch)	+	Supported through the plan type system decision (based on data) and the plan type user decision (based on a user selection on the wavefront). Moreover, it is possible to specify guards on the arcs in the static, dynamic and sequential subplans. When a guard evaluates to true the subsequent activity is enabled otherwise it is skipped (status “refused”).
5 (simple-m)	+	Supported by the end nodes of the plan type system decision and the plan type user decision. Multiple incoming arcs in static, dynamic and sequential subplans can be used to merge flows.
6 (m-choice)	+	Supported by guards on arcs in static, dynamic and sequential subplans.
7 (s-sync-m)	+	Supported inside static, dynamic and sequential subplans. Each plan model is a directed acyclic graph of nodes representing various plan elements and actions. Nodes with multiple incoming arcs wait for their predecessors to be completed or skipped (called “refused”). If all preceding nodes are skipped or all incoming arcs have a guard evaluating to false, a node is skipped. Otherwise normal processing is resumed. Note that the approach is similar to passing true and false tokens. True tokens correspond to arcs that evaluate to true and get triggered by completed nodes. False tokens correspond to arcs that evaluate to false or arcs that are skipped (i.e., the preceding node is “refused”). The join semantics is that if a node has at least one true token as input it becomes enabled. If all input tokens are false it is skipped (i.e., labeled as “refused”).
8 (multi-m)	+/-	It is possible to have multiple concurrent threads using dynamic subplans. Therefore, there is partial support for the pattern. However, since plans are highly structured, it is not possible to have an AND-split/XOR-join type of situation, i.e., the models are essentially “safe” (1-bounded).

<b>Pattern</b>	<b>Score</b>	<b>Motivation</b>
9 (s-disc)	–	Not supported. Dynamic subplans can have an auto complete condition however this is only evaluated when all subplans have completed.
10 (arb-c)	–	Not supported. In fact there are no loops and the language is block structured. Each plan model is a directed acyclic graph of nodes representing various plan elements and actions. Iteration is achieved through the sequential subplan or the redo role.
11 (impl-t)	+	Supported, plans can have multiple end nodes. Only if all are completed (or refused), the plan is completed.
12 (mi-no-s)	+	Directly supported through dynamic subplans. The dynamic subplan can be put into another plan such that subsequent plan elements do not have to wait for the completion of the plan.
13 (mi-dt)	+	Directly supported through dynamic subplans. For a dynamic plan, the minimum and maximum number of instances can be selected and the actual number of instances may be based on some expression. This expression can be a constant, thus realizing the pattern. Multiple instance data can be passed and accessed through a so-called dynamic array.
14 (mi-rt)	+	Directly supported through dynamic subplans. One can specify a variable number of instances (see above).
15 (mi-no)	+	Directly supported through dynamic subplans. It is possible to create new instances during execution. There is a setting “User may create instances”.

Pattern	Score	Motivation
16 (def-c)	+	Although there is no explicit notion of state, there are different ways of realizing this pattern: (1) The plan type user decision (based on user selection on the wavefront) can make the implicit choice in some cases. (2) The plan type system decision can make the implicit choice in other cases. Note that a system decision blocks until at least one of its conditions is true. As a consequence, race conditions based on time or external triggers are possible. In the latter case, triggering is handled through data-dependencies rather than explicit control-flow dependencies. (3) Yet another way to use a specific type of deferred choice is by using guards on arcs in a plan model (i.e., inside a static, dynamic or sequential plan) that evaluate to NIL. In this case processing stops until the guard evaluates to true or false. A guard evaluates to NIL if e.g. it contains an undefined variable. Moreover, using the operator “HasValue” this can be exploited. This way a deferred choice may be implemented via data. Note that data can be set from outside the process (e.g., based on a trigger).
17 (int-par)	+/-	Due to the case metaphor there is only one actor working on the case. Therefore, there is no true concurrency and any parallel routing is interleaved. Since true concurrency is not possible, a partial support rating is given.
18 (milest)	+/-	There is no direct support for milestones since there is no notion of state. However, in all situations, data dependencies can be used to emulate the construct. Simply introduce a data element (i.e., place in Petri-net terms) for each state for which a milestone needs to be specified.
19 (can-a)	+/-	It is possible to skip or redo activities. However, it is not possible to withdraw an activity in one branch triggered by an activity in another branch. Skip and redo are explicit user actions. Therefore, they provide only partial support.
20 (can-c)	+/-	It is possible to skip or redo an entire plan. However, skip and redo actions are always explicit user actions. Therefore, they provide only partial support. Note that by defining a data element named cancel and using this data element as a precondition for every activity in the flow it is possible to block a case. Although this is an elegant solution, it is still considered to be indirect.

Pattern	Score	Motivation
21 (str-l)	+	Iteration can be achieved through the use of the sequential plan construct.
22 (recur)	–	Not supported.
23 (t-trig)	–	Triggers can be modeled by setting data elements. There are various ways to wait for data, e.g., using guards on arcs or mandatory data elements of a milestone. The data elements can be considered as persistent. To model transient triggers one needs to reset the data value shortly after it has been set.
24 (p-trig)	+	Triggers can be modeled by setting data elements. There are various ways to wait for data, e.g., using guards on arcs or mandatory data elements of a milestone. This naturally corresponds to persistent triggers.
25 (can-r)	–	Not supported. There is no integrated means of cancelling a group of plans or plan elements.
26 (can-mi)	–	No direct means of cancelling a dynamic subplan without triggering the next activity.
27 (comp-mi)	+/–	A dynamic subplan can have an auto-complete condition can be specified for the subplan based on a variety of conditions however it only completes when all instances have completed. The use of deadlines on a dynamic subplan results in both the remaining instances and all subsequent activities in the process instance being force completed when the deadline is reached.
28 (b-disc)	–	Not supported. Note that the auto complete condition of a dynamic subplan cannot be used to continue processing at a higher level while blocking the dynamic subplan until all instances complete.
29 (c-disc)	–	Not supported. Dynamic subplans can have an auto complete condition however this is only evaluated when all subplans have completed.
30 (s-pjoin)	–	Not supported. Dynamic subplans can have an auto complete condition however this is only evaluated when all subplans have completed.
31 (b-pjoin)	–	Not supported (see above: the remaining subplan activities are forced to complete).
32 (c-pjoin)	–	Not supported. Dynamic subplans can have an auto complete condition however this is only evaluated when all subplans have completed.
33 (g-and-join)	–	Not supported. Note that FLOWer models are “safe” (1-bounded).
34 (st-pjoin-mi)	–	Not supported. Dynamic subplans can have an auto complete condition however this is only evaluated when all subplans have completed.

Pattern	Score	Motivation
35 (c-pjoin-mi)	–	Not supported. Dynamic subplans can have an auto complete condition however this is only evaluated when all subplans have completed.
36 (dyn-pjoin-mi)	-	Not supported. Dynamic subplans can have an auto complete condition however this is only evaluated when all subplans have completed.
37 (a-sync-m)	+	Supported inside the static, dynamic and sequential subplans. Each plan model is a directed acyclic graph of nodes representing various plan elements and actions. Nodes with multiple incoming arcs wait for the predecessors to be completed or skipped (called “re-fused”). If all preceding nodes are skipped or all incoming arcs have a guard evaluating to false, a node is skipped. Otherwise normal processing is resumed.
38 (g-sync-m)	–	Not supported because each plan model need to correspond to an acyclic graph.
39 (crit-sec)	+/-	Not directly supported. However, data elements can acts as semaphores. There are no concurrency problems because of the write-lock on cases.
40 (int-rout)	+/-	As for Interleaved Parallel Routing, the case metaphor allows a user to work on any of its constituent activities without regard to their overall sequence. A case is completed when all of its activities have been completed. Although there is no direct means of ensuring that activities are only undertaken one at a time, the fact that they are all undertaken by the same user obviates any potential for concurrency and ensures that they are interleaved.
41 (tm)	–	Not supported. The case metaphor prevents any possibility of multiple threads of execution.
42 (ts)	–	Not supported. The case metaphor prevents any possibility of multiple threads of execution.
43 (exp-t)	–	Not supported. Plans complete when all end nodes have completed.

## D COSA

This evaluation is based on COSA 5.1.

Pattern	Score	Motivation
1 (seq)	+	Directly supported by arcs connecting activities.
2 (par-spl)	+	Supported by multiple outgoing arcs from an activity. None of the arcs have transition conditions specified.
3 (synch)	+	Supported by multiple incoming arcs to an activity. None of the arcs have transition conditions specified.
4 (ex-ch)	+	Supported by multiple outgoing arcs from an activity. All of the arcs have disjoint transition conditions specified.
5 (simple-m)	+	Supported by multiple incoming arcs to a place.
6 (m-choice)	+	Supported by multiple outgoing arcs from an activity. The arcs may have (possibly overlapping) transition conditions specified.
7 (s-sync-m)	–	Processes are not inherently structured.
8 (multi-m)	+ / –	Only safe Petri net diagrams can be used.
9 (s-disc)	–	The discriminator can be modelled by using true conditions in input arcs and extending the network. Unfortunately, the resulting diagram is too complex.
10 (arb-c)	+	Supported. Any graph structure is allowed.
11 (impl-t)	–	Not supported, explicit termination is needed.
12 (mi-no-s)	+	COSA has a three level workflow model, i.e., workflow, flow, and activity. Flows (i.e., workflow instances) can be grouped in one workflow and share information. This combined with a trigger mechanism to create new flows is a possible solution where folders are used to facilitate shared access to common data.
13 (mi-dt)	–	There is no means of denoting that an activity should be executed multiple times.
14 (mi-rt)	–	There is no means of denoting that an activity should be executed multiple times.
15 (mi-no)	–	There is no means of denoting that an activity should be executed multiple times.
16 (def-c)	+	Supported by multiple outgoing arcs from a place.
17 (int-par)	+	Directly supported through places and also an optional setting of the workflow engine.
18 (milest)	+	Directly supported through places.
19 (can-a)	+	Supported by removing tokens from input places.
20 (can-c)	–	Only supported through an API call.
21 (str-l)	–	There is no means of specifying repeated execution of an activity or set of activities.
22 (recur)	+	Recursive definition of process models can be achieved using triggers or the <code>activ_run</code> tool agent.

Pattern	Score	Motivation
23 (t-trig)	+	Supported through trigger construct in synchronous mode.
24 (p-trig)	+	Supported through trigger construct in asynchronous mode.
25 (can-r)	+/-	Achievable by specifying a shadow cancellation activity for each activity in the cancellation region although the overall diagram is likely to be intractable for anything other than simple process models.
26 (can-mi)	-	Multiple instance activities are not supported.
26 (comp-mi)	-	Multiple instance activities are not supported.
28 (b-disc)	-	There is no modelling construct that directly corresponds to this pattern and although the behaviour can be indirectly achieved, the process model required to do so is too complex.
29 (c-disc)	-	Similar to WCP-28 (b-disc), no direct support.
30 (s-pjoin)	-	There is no modelling construct that directly corresponds to this pattern and although the behaviour can be indirectly achieved, the process model required to do so is too complex.
31 (b-pjoin)	-	Similar to WCP-30 (s-pjoin), no direct support.
32 (c-pjoin)	-	Similar to WCP-30 (s-pjoin), no direct support.
33 (g-and-join)	-	Only safe Petri net diagrams can be used.
34 (st-pjoin-mi)	-	Multiple instance activities are not supported.
35 (c-pjoin-mi)	-	Multiple instance activities are not supported.
36 (dyn-pjoin-mi)	-	Multiple instance activities are not supported.
37 (a-sync-m)	+	The condition on an input arc to an activity can specified such that it will not be considered when the join condition for the activity is evaluated if the branch to which it belongs is not live.
38 (g-sync-m)	-	No ability to determine when an OR-join should fire based on a complete evaluation of the overall state of the process instance.
39 (crit-sec)	+	Supported through the use of a mutex place to prevent concurrent access to the critical section.
40 (int-rout)	+	Supported through the use of a mutex place to prevent nominated activities from executing concurrently.
41 (tm)	-	No ability to merge threads as the process is inherently safe.
42 (ts)	-	No ability to merge threads as the process is inherently safe.
43 (exp-t)	+	Directly supported, a process instance completes when an end activity is reached.



## E iPlanet

This evaluation is based on iPlanet Integration Server 3.0.

Pattern	Score	Motivation
1 (seq)	+	Directly supported by the use of activity routers.
2 (par-spl)	+	Supported by multiple outgoing routers from an activity.
3 (synch)	+	Supported by specifying a trigger condition for an activity with multiple incoming routers that only fires when all routers are activated.
4 (ex-ch)	+	Supported by using multiple outgoing routers from an activity with disjoint router enabling conditions.
5 (simple-m)	+	Supported by specifying a trigger condition for an activity with multiple incoming routers that fires when any incoming router is activated.
6 (m-choice)	+	Supported by multiple outgoing routers from an activity, each with specific (and possibly overlapping) enabling conditions.
7 (s-sync-m)	–	Not supported. Process models are not necessarily structured.
8 (multi-m)	+	Supported by specifying a trigger condition for an activity with multiple incoming routers that fires when any incoming router is activated.
9 (s-disc)	+	Supported through the use of a customised trigger condition for an activity that only fires when the first incoming router is activated.
10 (arb-c)	+	Arbitrary loop structures are able to be represented.
11 (impl-t)	–	There is a designated last activity which causes process termination.
12 (mi-no-s)	+	Supported via asynchronous subprocess activities.
13 (mi-dt)	–	Not supported. No means of designating that multiple instances of an activity are required.
14 (mi-rt)	–	Not supported. No means of designating that multiple instances of an activity are required.
15 (mi-no)	–	Not supported. No means of designating that multiple instances of an activity are required.
16 (def-c)	–	Not supported. No concept of state.
17 (int-par)	–	There is no way to interleave activities without actually enumerating all possible execution sequences within the process model and selecting one of them at runtime.
18 (milest)	–	Not supported. No concept of state.
19 (can-a)	+	Supported via the AbortActivity method.
20 (can-c)	–	Not supported. There is no means of terminating a process instance.

Pattern	Score	Motivation
21 (str-l)	+	Supported through the use of process variables in conjunction with routers.
22 (recur)	+	Supported via synchronous subprocess activities.
23 (t-trig)	–	No trigger support.
24 (p-trig)	–	No trigger support.
25 (can-r)	–	No means of cancelling a region of a process model.
26 (can-mi)	–	No support for multiple instance activities.
27 (comp-mi)	–	No support for multiple instance activities.
28 (b-disc)	–	Not supported. No ability to block activity triggerings.
29 (c-disc)	–	Not supported. No ability to cancel portions of a process model.
30 (s-pjoin)	+	Supported through the use of a customised trigger condition for an activity that only fires when the Nth incoming router is activated.
31 (b-pjoin)	–	Not supported. No ability to block activity triggerings.
32 (c-pjoin)	–	Not supported. No ability to cancel portions of a process model.
33 (g-and-join)	–	Not supported. No ability to buffer activity triggers.
34 (st-pjoin-mi)	–	No support for multiple instance activities.
35 (c-pjoin-mi)	–	No support for multiple instance activities.
36 (dyn-pjoin-mi)	–	No support for multiple instance activities.
37 (a-sync-m)	–	No means of providing information to an OR-join to enable local determination of when it should fire.
38 (g-sync-m)	–	No ability to assess when an OR-join should fire through analysis of current/future state.
39 (crit-sec)	–	Not supported. Although custom router conditions could be specified that access a mutex variable to determine when an activity can proceed, there is no means of managing concurrent access to the variable.
40 (int-rout)	–	There is no way to interleave activities without actually enumerating all possible execution sequences within the process model and selecting one of them at runtime.
41 (tm)	–	No ability to coalesce threads of control from independent sub-process activities.
42 (ts)	–	No ability to coalesce threads of control from independent sub-process activities.
43 (exp-t)	+	Directly supported. There is a designated last activity which causes process termination.

## F SAP Workflow

This evaluation is based on SAP Workflow version 4.6c.

Pattern	Score	Motivation
1 (seq)	+	Directly supported. In SAP one can connect activities using arcs, thus creating a sequence.
2 (par-spl)	+	Directly supported. SAP allows for structured parallel processes, using the fork construct one can create multiple parallel branches. Since there has to be a one-to-one correspondence between splits and joins, some parallel processes need to be modified to make them structured.
3 (synch)	+	Directly supported via the fork construct. However, as indicated before, there has to be a one-to-one correspondence between split and join.
4 (ex-ch)	+	Directly supported through three constructs: (1) the condition step type, (2) the multiple condition step type, (3) a choice directly following an activity. The condition step type can only be used for binary conditions based on a single Boolean expression. The multiple condition step type can be used to select from more than two alternatives. There is default branch that is selected if none of the other branches can be taken. The choice directly following an activity is similar to the multiple condition step type but is not shown graphically in the workflow model. As for the parallel split and synchronization constructs, each split needs to correspond to a join, i.e. again only block structured models are supported.
5 (simple-m)	+	Directly supported as indicated before. However, there has to be a one-to-one correspondence between splits and joins.
6 (m-choice)	–	Not supported. Although there are three constructs to model choices, it is not possible to select multiple exits for a choice (other than the fork). Hence a multi-choice requires a combination of fork and condition constructs.
7 (s-sync-m)	–	Not supported for two reasons. First of all, it is not possible to create optional parallel branches other than explicitly skipping the branches that are not selected. Second, the join construct of a fork is unaware of the number of truly active branches. Therefore, any synchronizing merge needs to be rewritten as a mix of forks and conditions.

Pattern	Score	Motivation
8 (multi-m)	–	Not supported because the block structure of SAP Workflow forces the model to be “safe” (in Petri-net terms), i.e. it is not possible to enable or activate an activity in parallel with itself.
9 (s-disc)	+/–	This is supported by the fork construct which allows for the specification of the number of branches that needs to be complete or some other end condition, i.e. a fork can start 10 branches and terminate 9 branches upon completion of the first branch. Note that all remaining branches are cancelled hence this construct only achieves partial support.
10 (arb-c)	–	SAP workflow models are inherently block structured, i.e. any split corresponds to a join and only structured loops (while/until loops) are possible using the loop step type.
11 (impl-t)	–	Not supported because processes are block structured with a single start and end node.
12 (mi-no-s)	+/–	The pattern is partially supported through data objects and events triggering workflows, i.e. a state change of an object may trigger a workflow. By changing states in a loop it is possible to trigger multiple instances. Note that the newly created instances run in a distinct process instances that has no relation to the instance of the activity activating them. Hence only partial support is given.
13 (mi-dt)	+	This pattern is supported in two ways: (1) indirectly via the use of loop construct together with a counter variable indicating how many instances are to be synchronized by the subsequent join and (2) via “dynamic processing with a multi-line container element”. Based on a multi-line data element (i.e., a table), an activity or sub-process is executed as many times as there are lines in the multi-line data element (i.e. the number of rows). There is a maximum of 99 instances. Through this “dynamic processing with a multi-line container element” dynamic support is achieved.
14 (mi-rt)	+	Supported directly via the “dynamic processing with a multi-line container element” as indicated for the previous pattern. The number of rows is only relevant at the moment of activation and does not need to be fixed before.

Pattern	Score	Motivation
15 (mi-no)	–	Not supported. The only way to realize this is through the use of loop construct together with a counter variable indicating how many instances are to be synchronized by the subsequent join. This number can be modified at run-time. However, the designer has to do the book-keeping to link events to activities.
16 (def-c)	–	Not supported. It can only be realized by starting multiple branches in parallel using the fork construct and then commencing each branch with a “wait for event” step. Once the first event completes and the subsequent activity completes, the other branches can be cancelled by setting the required number of complete branches to 1 in the join node of the fork. However, when using this approach it is still possible that multiple activities run in parallel. Clearly, this is not sufficient support for this pattern.
17 (int-par)	–	Not supported, the concept of a state is completely missing and there seems no way to influence the fork construct to only allow for interleavings (and not true concurrency).
18 (milest)	–	Not supported, because the concept of a state is completely missing.
19 (can-a)	+	Supported through the use of the “process control” step. A process control step can be configured such that it forces another work item of the same workflow into the status “logically deleted”, i.e. another activity is cancelled. This completes this other work item and subsequent steps of this work item are not executed. To indicate the activity to be cancelled, you specify the node number of the corresponding step.
20 (can-c)	+	Also supported through the use of the “process control” step. A process control step can be set to “terminate workflow” which forces all work items of the same workflow into the status “logically deleted” and terminates the current process instance by setting it to status completed. If the terminated workflow was a sub-workflow of a superior workflow, the system executes the binding between the container of the sub-workflow and the container of the superior workflow in accordance with the definition, and continues the superior workflow.

Pattern	Score	Motivation
21 (str-l)	+	This pattern is supported through the loop construct. There are two types of loops: (1) the until loop and (2) the while loop. The until loop has a “body” that is first executed. After executing the body a Boolean condition is evaluated. If it evaluates to true, the process continues with the next step. Otherwise, the process the “loopback body” is executed and then the process is repeated from the beginning. The while loop is similar to the until loop, however, the “body” is always empty. Therefore, the “loopback body” is the only part inside the loop and this part is executed zero or more times.
22 (recur)	+	A “multistep task” (i.e. an activity that is decomposed into a sub-workflow) contains a reference to a workflow definition hence a workflow could be decomposed into itself.
23 (t-trig)	+	SAP offer a “wait for event” step that can be used to wait for an event. This step has different settings. The standard mechanism is that events are not queued. There is an event queue but this is just there for performance reasons.
24 (p-trig)	+	It is possible to use the “wait for event” step with the setting “event via workflow”. When waiting for an event via the workflow, the event is initially received and temporarily saved by the workflow. Once the wait step has been activated, the event is forwarded to the wait step.
25 (can-r)	–	The process control step can terminate/cancel a specific activity of the whole process instance. It is not possible to cancel everything in a region other than by enumerating the region using a sequence of process control steps.
26 (can-mi)	+	Via “dynamic processing with a multi-line container element” it is possible to create an instance for each row in a table. The termination of the parent activity can be passed on to the child objects.
27 (comp-mi)	–	Not supported. An activity set to “logically deleted” is recursively scanned for items that do not yet have the status completed. These work items are then also set to the status “logically deleted”. Hence it is not possible to complete the multiple instances still running.
28 (b-disc)	–	The block structured nature of SAP workflow does not allow for concurrent execution threads within the same instance. Hence a discriminator cannot be activated multiple times.

Pattern	Score	Motivation
29 (c-disc)	+	This pattern is supported by the fork construct which allows for the specification of the number of branches that needs to complete. This can be set 1 one thus resulting in a discriminator. When completing the first branch all remaining branches are cancelled.
30 (s-pjoin)	+/-	SAP workflow only supports structured workflows. In the case of the partial join, this is supported by a fork that can start M branches in parallel and the fork completes after the completion of the first N branches. The remaining branches are cancelled hence this construct only achieves partial support.
31 (b-pjoin)	-	Not supported because of the structured/safe nature of SAP workflow.
32 (c-pjoin)	+	SAP workflow only supports structured workflows. In the case of the partial join, this is supported by a fork that can start M branches in parallel and the fork completes after the completion of N of these branches. The remaining branches are cancelled.
33 (g-and-join)	-	Not supported because of the structured/safe nature of SAP workflow.
34 (st-pjoin-mi)	-	Not supported. Via “dynamic processing with a multi-line container element” it is possible to create an instance for each row in a table. However, these instances are supposed to complete and there is no setting for a threshold value.
35 (st-pjoin-mi-fc)	-	Not supported. Via “dynamic processing with a multi-line container element” it is possible to create an instance for each row in a table. However, these instances are supposed to complete.
36 (dyn-pjoin-mi)	-	Not supported. Note that “dynamic processing with a multi-line container element” does not allow for dynamic changes of the number of instances.
37 (a-sync-m)	-	Not supported for two reasons. First of all, it is not possible to create optional parallel branches other than explicitly skipping the branches that are not selected. Second, the join construct of a fork is unaware of the number of truly active branches. Therefore, any synchronizing merge needs to be rewritten as a mix of forks and conditions.
38 (g-sync-m)	-	As indicated above none of the variants of the synchronizing merge are supported.
39 (crit-sec)	-	Not supported. There are no mechanisms other than events and cancellations to support interactions between parallel branches.

<b>Pattern</b>	<b>Score</b>	<b>Motivation</b>
40 (int-rout)	–	Not supported. There are no mechanisms other than events and cancellations to support interactions between parallel branches.
41 (tm)	–	Not supported because of the structured/safe nature of SAP workflow.
42 (ts)	–	Not supported because of the structured/safe nature of SAP workflow.
43 (expl-t)	+	Directly supported. Processes are block structured with a single start and end node.



## G FileNet

This evaluation is based on FileNet P8 BPM Suite version 3.5.

Pattern	Score	Motivation
1 (seq)	+	Directly supported by means of steps connected with unconditional routes.
2 (par-spl)	+	Directly supported by a step where all outgoing routes are unconditional.
3 (synch)	+	Directly supported by a component the incoming routing info of which is set to “collector step”.
4 (ex-ch)	+	Directly supported by a step with multiple outgoing routes (take route of the first true condition). Each of the routes must have a condition associated with it, all defined conditions must be exclusive. If several conditions are satisfied, the first specified in the lexical order is selected.
5 (simple-m)	+	Directly supported by a step (which is not a collector step).
6 (m-choice)	+	Directly supported by a step, which takes routes of all true conditions. Requires structure followed by a collector step.
7 (s-sync-m)	+	Directly supported by a collector step used a join in the structure.
8 (multi-m)	+	Directly supported, workflow waits for all active steps to finish.
9 (s-disc)	–	Not supported: no means for resetting are available.
10 (arb-c)	+	Directly supported: allows to specify cycles with multiple entry and exit points.
11 (impl-t)	+	Directly supported. Allows for multiple end-points, however workflow terminates after all steps have finished.
12 (mi-no-s)	+	Supported via invoke in the loop.
13 (mi-dt)	–	Not supported.
14 (mi-rt)	–	Not supported.
15 (mi-no)	–	Not supported.
16 (def-c)	+/–	Partially supported. It is possible to withdraw a timer, but not possible to withdraw an activity.
17 (int-par)	–	Not supported.

Pattern	Score	Motivation
18 (milest)	–	Not supported: although Filenet has a concept of milestone, it refers to the following: To track the progress of a workflow, the workflow author can define key points (milestones) in the workflow. On the workflow map, a milestone can be placed either before or after a General step, or after the Launch step. When the running workflow reaches a milestone, an author-specified message is written to a log file and, depending on its author-specified level (1 to 99), the milestone displays for workflow participants, trackers, and the user who launched the workflow. The Milestones page displays a list of milestones that have been reached for a workflow. You can only access this page from the email message sent to the workflow originator when the milestone is reached.
19 (can-a)	+	Directly supported via <Terminate Branch> step.
20 (can-c)	+	Directly supported via <Terminate Process> step. Furthermore, if none of the conditions could be satisfied, the workflow terminates.
21 (str-l)	+	Directly supported.
22 (recur)	–	Not supported.
23 (t-trig)	–	Not supported.
24 (p-trig)	+	Directly supported via <WaitForCondition> and <Receive> steps.
25 (can-r)		Not supported.
26 (can-mi)	–	No inherent support for multiple instance activities.
27 (can-mi)	–	No inherent support for multiple instance activities.
28 (b-disc)	–	Not supported.
29 (c-disc)	–	Not supported.
30 (s-pjoin)	–	Not supported.
31 (b-pjoin)	–	Not supported.
32 (c-pjoin)	–	Not supported.
33 (g-and-join)	+	Supported by a collector step.
34 (st-pjoin-mi)	–	Not supported.
35 (c-pjoin-mi)	–	Not supported.
36 (dyn-pjoin-mi)	–	Not supported.
37 (a-sync-m)	–	Not supported.
38 (g-sync-m)	+	Supported by a collector step.
39 (crit-sec)	–	Not supported.
40 (int-rout)	–	Not supported.
41 (tm)	–	Not supported.
42 (ts)	–	Not supported.
43 (exp-t)	–	Not supported. Workflow terminates after all steps have finished.

## H BPEL

This evaluation is based on BPEL4WS version 1.1.

Pattern	Score	Motivation
1 (seq)	+	Supported by <sequence> or links within the <flow> construct.
2 (par-spl)	+	Supported by <flow> construct.
3 (synch)	+	Supported by <flow> construct.
4 (ex-ch)	+	Supported by <switch> or links within the <flow> construct.
5 (simple-m)	+	Supported by <switch> or links within the <flow> construct.
6 (m-choice)	+	Supported by links within the <flow> construct.
7 (s-sync-m)	+	Supported by links within the <flow> construct.
8 (multi-m)	–	Not supported. The language is block structured and it is not possible for two threads of execution to run through the same path in a single process instance.
9 (s-disc)	–	Not supported. There is no dedicated language construct and links cannot be used in conjunction with an OR joinCondition as the join requires the status of all incoming links to be known before evaluation, not just the identification of the first positive link.
10 (arb-c)	–	Not supported. The language is block structured and cannot capture unstructured cycles.
11 (impl-t)	+	Directly supported for the <flow> construct. For other construct, each branch must be ended with an end event node.
12 (mi-no-s)	+	Supported by <invoke> statement within <while> loop.
13 (mi-dt)	–	Not supported. No direct means of denoting multiple activity instances are required.
14 (mi-rt)	–	Not supported. No direct means of denoting multiple activity instances are required.
15 (mi-no)	–	Not supported. No direct means of denoting multiple activity instances are required.
16 (def-c)	+	Supported by the <pick> construct
17 (int-par)	+/-	Indirectly supported via serializable scopes but limited to activities within the same scope.
18 (milest)	–	Indirectly achievable by using a <pick> construct within a <while> loop which can only be executed once but solution is overly complex.
19 (can-a)	+	Supported through fault and compensation handlers.
20 (can-c)	+	Supported by <terminate> construct.
21 (str-l)	+	While loops are directly supported.
22 (recur)	–	Not supported. Recursive composition is possible on an external basis using the <invoke> construct against web services but there is no internal support.

Pattern	Score	Motivation
23 (t-trig)	–	Not supported. Messages are durable in form.
24 (p-trig)	+	Supported by <pick> construct waiting on specific message type.
25 (can-r)	+/-	No means of cancelling arbitrary groups of activities although activities within the same scope can be cancelled.
26 (can-mi)	–	No support for multiple activity instances.
27 (comp-mi)	–	No support for multiple activity instances.
28 (b-disc)	–	Not supported. There is no dedicated language construct and links cannot be used in conjunction with an OR joinCondition as the join requires the status of all incoming links to be known before evaluation, not just the identification of the first positive link.
29 (c-disc)	–	Not supported. As for WCP-28.
30 (s-pjoin)	–	Not supported. Similar to the discriminator, there is no dedicated language construct and links cannot be used in conjunction with an OR joinCondition as the join requires the status of all incoming links to be known before evaluation, not just the identification of the first N positive links.
31 (b-pjoin)	–	Not supported. As for WCP-30.
32 (c-pjoin)	–	Not supported. As for WCP-30.
33 (g-and-join)	–	Not supported. There is no notion of multiple execution threads through a single path in a process instance.
34 (st-pjoin-mi)	–	No support for multiple activity instances.
35 (st-pjoin-mi-fc)	–	No support for multiple activity instances.
36 (dyn-pjoin-mi)	–	No support for multiple activity instances.
37 (a-sync-m)	+	Supported by links within a <flow> construct.
38 (g-sync-m)	–	Not supported. Process models are always block structured and OR-joins always operate within a <flow> construct.
39 (crit-sec)	+	Supported by serializable scopes.
40 (int-rout)	+	Supported by serializable scopes.
41 (tm)	+/-	The correlation facility for invoked activities provides the basis for coalescing distinct threads of control but programmatic extensions are necessary to keep track of when the merge should occur.
42 (ts)	+/-	Achievable through the use of the <invoke> construct in conjunction with the correlation facility but programmatic extensions are necessary if subsequent thread merges are required.
43 (exp-t)	–	Process instances complete when no activity instances remain to complete.

## I WebSphere BPEL

This evaluation is based on WebSphere Integration Developer v6.0, which is the development environment for the Business Process Choreographer (BPC) part of WebSphere Process Server v6.0.2. Throughout this document, we informally refer to this offering as WebSphere BPEL.

Pattern	Score	Motivation
1 (seq)	+	Supported by the <sequence> activity.
2 (par-spl)	+	Supported by the <flow> activity.
3 (synch)	+	Supported by the <flow> activity.
4 (ex-ch)	+	Supported by the <switch> activity or links within a <flow> activity.
5 (simple-m)	+	Supported by the <switch> activity or links within a <flow> activity.
6 (m-choice)	+	Supported by the <switch> activity or links within a <flow> activity.
7 (s-sync-m)	+	Supported by links within the <flow> activity.
8 (multi-m)	–	Not supported. The language is block structured and it is not possible for two threads of execution to run through the same path in a single process instance.
9 (s-disc)	–	Not supported. There is no dedicated language construct and links cannot be used in conjunction with an OR joinCondition as the join requires the status of all incoming links to be known before evaluation, not just the identification of the first positive link.
10 (arb-c)	–	Not supported. The language is block structured and cannot capture unstructured cycles.
11 (impl-t)	+	Directly supported for the <flow> activity. For other constructs, each branch must be ended with a <terminate> activity.
12 (mi-no-s)	+	Supported by the <invoke> activity within a <while> loop.
13 (mi-dt)	–	Not supported. No direct means of denoting multiple activity instances are required.
14 (mi-rt)	–	Not supported. No direct means of denoting multiple activity instances are required.
15 (mi-no)	–	Not supported. No direct means of denoting multiple activity instances are required.
16 (def-c)	+	Supported by the <pick> activity.
17 (int-par)	+/-	Indirectly achievable via serializable scopes but this limits the activities to those within the same scope.
18 (milest)	–	Indirectly achievable by using a <pick> activity within a <while> loop which can only be executed once but solution is overly complex.
19 (can-a)	+	Supported through fault and compensation handlers.
20 (can-c)	+	Supported by the <terminate> activity.

Pattern	Score	Motivation
21 (str-l)	+	While loops are directly supported.
22 (recur)	–	Not supported. Recursive composition is possible on an external basis using the <invoke> construct against web services but there is no internal support.
23 (t-trig)	–	Not supported. Messages are durable in form.
24 (p-trig)	+	Supported by the <pick> activity waiting on specific message type.
25 (can-r)	+/-	No means of cancelling arbitrary groups of activities although activities within the same scope can be cancelled.
26 (can-mi)	–	No support for multiple activity instances.
27 (comp-mi)	–	No support for multiple activity instances.
28 (b-disc)	–	Not supported. There is no dedicated language construct and links cannot be used in conjunction with an OR joinCondition as the join requires the status of all incoming links to be known before evaluation, not just the identification of the first positive link.
29 (c-disc)	–	Not supported. As for WCP-28.
30 (s-pjoin)	–	Not supported. Similar to the discriminator, there is no dedicated language construct and links cannot be used in conjunction with an OR joinCondition as the join requires the status of all incoming links to be known before evaluation, not just the identification of the first N positive links.
31 (b-pjoin)	–	Not supported. As for WCP-30.
32 (c-pjoin)	–	Not supported. As for WCP-30.
33 (g-and-join)	–	Not supported. There is no notion of multiple execution threads through a single path in a process instance.
34 (st-pjoin-mi)	–	No support for multiple activity instances.
35 (st-pjoin-mi-fc)	–	No support for multiple activity instances.
36 (dyn-pjoin-mi)	–	No support for multiple activity instances.
37 (a-sync-m)	+	Supported by links within a <flow> construct.
38 (g-sync-m)	–	Not supported. Process models are always block structured and OR-joins always operate within a <flow> construct.
39 (crit-sec)	+	Supported by serializable scopes.
40 (int-rout)	+	Supported by serializable scopes.
41 (tm)	+/-	The correlation facility for invoked activities provides the basis for coalescing distinct threads of control but programmatic extensions are necessary to keep track of when the merge should occur.
42 (ts)	+/-	Achievable through the use of the <invoke> construct in conjunction with the correlation facility but programmatic extensions are necessary if subsequent thread merges are required.

Pattern	Score	Motivation
43 (exp-t)	–	Process instances complete when no activity instances remain to complete.

## J Oracle BPEL

This evaluation is based on Oracle BPEL v10.1.2.

Pattern	Score	Motivation
1 (seq)	+	Supported by the <sequence> construct or links within the <flow> construct.
2 (par-spl)	+	Supported by the <flow> construct.
3 (synch)	+	Supported by the <flow> construct.
4 (ex-ch)	+	Supported by the <switch> construct or links within a <flow> construct.
5 (simple-m)	+	Supported by the <switch> construct or links within a <flow> construct.
6 (m-choice)	+	Supported by links within a <flow> construct.
7 (s-sync-m)	+	Supported by links within a <flow> construct.
8 (multi-m)	–	Not supported. The language is block structured and it is not possible for two threads of execution to run through the same path in a single process instance.
9 (s-disc)	–	Not supported. There is no dedicated language construct and links cannot be used in conjunction with an OR joinCondition as the join requires the status of all incoming links to be known before evaluation, not just the identification of the first positive link.
10 (arb-c)	–	Not supported. The language is block structured and cannot capture unstructured cycles.
11 (impl-t)	+	Directly supported for the <flow> construct. For other constructs, each branch must be ended with a <terminate> construct.
12 (mi-no-s)	+	Supported by the <invoke> construct within a <while> loop.
13 (mi-dt)	+	Supported through the use of the <flowN> construct which allows multiple concurrent instances of an activity to be created.
14 (mi-rt)	+	Supported through the use of the <flowN> construct which allows multiple concurrent instances of an activity to be created.
15 (mi-no)	–	Not supported. No direct means of initiating additional instances of a multiple activity (e.g. as created by the <flowN> construct) is available.
16 (def-c)	+	Supported by the <pick> construct.
17 (int-par)	–	Supported by the BPEL spec but not implementable in Oracle BPEL.
18 (milest)	–	Indirectly achievable by using a <pick> activity within a <while> loop which can only be executed once but solution is overly complex.
19 (can-a)	+	Supported by associating a fault or compensation handler with an activity.
20 (can-c)	+	Supported by the <terminate> activity.



Pattern	Score	Motivation
21 (str-l)	+	While loops are directly supported.
22 (recur)	–	Not supported. Recursive composition is possible on an external basis using the <invoke> construct against web services but there is no internal support.
23 (t-trig)	–	Not supported. Messages are durable in form.
24 (p-trig)	+	Supported by the <pick> activity waiting on specific message type.
25 (can-r)	+/-	No means of cancelling arbitrary groups of activities although activities within the same scope can be cancelled.
26 (can-mi)	+	Supported for multiple activity instances in a <flowN> construct by associating it with a fault or compensation handler.
27 (comp-mi)	–	No support for multiple activity instances.
28 (b-disc)	–	Not supported. There is no dedicated language construct and links cannot be used in conjunction with an OR joinCondition as the join requires the status of all incoming links to be known before evaluation, not just the identification of the first positive link.
29 (c-disc)	–	Not supported. As for WCP-28.
30 (s-pjoin)	–	Not supported. Similar to the discriminator, there is no dedicated language construct and links cannot be used in conjunction with an OR joinCondition as the join requires the status of all incoming links to be known before evaluation, not just the identification of the first N positive links.
31 (b-pjoin)	–	Not supported. As for WCP-30.
32 (c-pjoin)	–	Not supported. As for WCP-30.
33 (g-and-join)	–	Not supported. There is no notion of multiple execution threads through a single path in a process instance.
34 (st-pjoin-mi)	–	No means of .
35 (st-pjoin-mi-fc)	–	No support for multiple activity instances.
36 (dyn-pjoin-mi)	–	No support for multiple activity instances.
37 (a-sync-m)	+	Supported by links within a <flow> construct.
38 (g-sync-m)	–	Not supported. Process models are always block structured and OR-joins always operate within a <flow> activity.
39 (crit-sec)	+	Supported by serializable scopes.
40 (int-rout)	–	Supported by the BPEL spec but not implementable in Oracle BPEL.
41 (tm)	+/-	The correlation facility for invoked activities provides the basis for coalescing distinct threads of control but programmatic extensions are necessary to keep track of when the merge should occur.

Pattern	Score	Motivation
42 (ts)	+/-	Achievable through the use of the <invoke> construct in conjunction with the correlation facility but programmatic extensions are necessary if subsequent thread merges are required.
43 (exp-t)	-	Process instances complete when no activity instances remain to complete.

## K BPMN

The evaluation is based on BPMN version 1.0.

Pattern	Score	Motivation
1 (seq)	+	Directly supported by linking activities with sequence flow arcs.
2 (par-spl)	+	Supported by AND-split gateway.
3 (synch)	+	Supported by AND-join gateway.
4 (ex-ch)	+	Supported by XOR-split gateway.
5 (simple-m)	+	Supported by XOR-join gateway.
6 (m-choice)	+	Supported in three distinct ways: via an implicit split with conditions on the arcs, an OR-split or a complex gateway.
7 (s-sync-m)	+	Supported through the OR-join gateway.
8 (multi-m)	+	Supported by XOR-join gateway.
9 (s-disc)	+/-	Although support for this pattern is referred to in the BPMN 1.0 specification, it is unclear how the IncomingCondition expression on the COMPLEX-join gateway is specified.
10 (arb-c)	+	Unstructured repetition can be directly supported.
11 (impl-t)	+	Supported by ending every thread with an End Event. When the last token generated by the Start Event is consumed, the process instance terminates.
12 (mi-no-s)	+	Supported via multiple instance task with MI Flow_Condition attribute set to none.
13 (mi-dt)	+	Supported via multiple instance task with MI Flow_Condition attribute set to all.
14 (mi-rt)	+	Supported via multiple instance task with MI.Condition attribute set at runtime to the actual number of instances required.
15 (mi-no)	-	Not supported. There is no means of adding further instances to a multiple instance task once started.
16 (def-c)	+	Supported via an event-based exclusive gateway followed by either intermediate events using message-based triggers or receive tasks.
17 (int-par)	-	Supported for simple tasks via an ad-hoc process but no support for interleaving groups or sequences of tasks.
18 (milest)	-	Not supported. No support for states.
19 (can-a)	+	Supported via an error type intermediate event trigger attached to the boundary of the activity to be cancelled.
20 (can-c)	+	Directly supported by including the entire process in a transaction. Triggering the cancel end event associated with the transaction will effectively terminate all activities associated with a process instance.

Pattern	Score	Motivation
21 (str-l)	+	Both while and repeat loops are directly supported by activity looping.
22 (recur)	–	Not supported. No means of specifying recursive composition with a process model.
23 (t-trig)	–	Not supported. Triggers are supported through durable messages.
24 (p-trig)	+	Supported through the use of message events.
25 (can-r)	+/-	Partially supported by enclosing the cancellation region in a sub-process and associating an error event with the sub-process to enable cancellation, however the cancellation region is restricted to a connected sub-graph of the overall process model.
26 (can-mi)	+	Achievable via a MI task which has an error type intermediate event trigger at the boundary. When the MI activity is to be withdrawn, a cancel event is triggered to terminate any remaining MI activities.
27 (comp-mi)	–	Not supported. No means of cancelling remaining MI activity instances.
28 (b-disc)	+/-	Although support for this pattern is referred to in the BPMN 1.0 specification, it is unclear how the IncomingCondition expression on the COMPLEX-join gateway is specified.
29 (c-disc)	+	Supported by including the incoming branches and the OR-join in a subprocess that passes control to the following activity once the first branch has completed as well as cancelling the remaining activities in the subprocess using an error type intermediate event.
30 (s-pjoin)	+/-	Although support for this pattern is referred to in the BPMN 1.0 specification, it is unclear how the IncomingCondition expression on the COMPLEX-join gateway is specified.
31 (b-pjoin)	+/-	Although support for this pattern is referred to in the BPMN 1.0 specification, it is unclear how the IncomingCondition expression on the COMPLEX-join gateway is specified.
32 (c-pjoin)	+/-	Although support for this pattern is referred to in the BPMN 1.0 specification, it is unclear how the IncomingCondition expression on the COMPLEX-join gateway is specified.
33 (g-and-join)	+	Supported by the AND-join gateway.

Pattern	Score	Motivation
34 (st-pjoin-mi)	+/-	Notionally supported via multiple instance task with MI Flow_Condition attribute set to complex where ComplexMI_FlowCondition is an expression that evaluates to true when exactly M instances have completed and passes on a single token to the following activity. However, it is unclear exactly how the ComplexMI_FlowCondition should be specified.
35 (c-pjoin-mi)	+/-	Notionally achievable via a MI task (as for WCP-34) which has an error type intermediate event trigger at the boundary. Immediately after the MI task is an activity that issues a cancel event to terminate any remaining MI activities. However the same issue arises as for WCP-34 in that it is unclear how the ComplexMI_FlowCondition should be specified.
36 (dyn-pjoin-mi)	-	There is no ability to dynamically add instances to a multiple instance activity.
37 (a-sync-m)	-	The OR-join gateway assumes it will be used in a structured context.
38 (g-sync-m)	-	Not supported. No means of assessing whether an OR-join gateway should fire based on a complete state analysis of the process instance.
39 (crit-sec)	-	Not supported. No means of limiting concurrent execution of a set of activities.
40 (int-rout)	+/-	Indirectly supported via an ad-hoc process containing all of the activities to be interleaved with AdHocOrdering set to sequential however it is unclear what the required AdHocCompletionCondition should be to ensure each activity is executed precisely once.
41 (tm)	+	Directly supported by setting the StartQuantity attribute on activities immediately following the MI activity.
42 (ts)	+	Directly supported by setting the Quantity attribute on the outgoing sequence flow from an activity.
43 (exp-t)	+	Supported via a terminate end event.

## L XPDL

This evaluation is based on XPDL version 2.0.

Pattern	Score	Motivation
1 (seq)	+	Supported by the transition construct.
2 (par-spl)	+	Supported by the AND-split construct.
3 (synch)	+	Supported by the AND-join construct.
4 (ex-ch)	+	Supported by the XOR-split construct.
5 (simple-m)	+	Supported by the XOR-join construct.
6 (m-choice)	+	Supported by the AND-split construct together with conditions on the outgoing transitions.
7 (s-sync-m)	+	Supported by the OR-join construct.
8 (multi-m)	+	Supported by the XOR-join construct.
9 (s-disc)	+/-	Although the COMPLEX-join gateway appears to offer support for this pattern, it is unclear how the IncomingCondition expression is specified.
10 (arb-c)	+	Unstructured repetition can be directly supported.
11 (impl-t)	+	Supported by ending every thread with an End Event. When the last token generated by the Start Event is consumed, the process instance terminates.
12 (mi-no-s)	+	Supported by spawning off activity instances in a loop.
13 (mi-dt)	+	Supported by the multi-instance loop construct.
14 (mi-rt)	+	Supported by the multi-instance loop construct where MI.Ordering = parallel.
15 (mi-no)	-	Not supported. There is no means of adding further instances to a multi-instance loop once started.
16 (def-c)	+	Supported by the XOREVENT-split construct.
17 (int-par)	-	Supported for single activities (but not sequences) by grouping them using the ActivitySet construct with the AdHoc attribute set.
18 (milest)	-	Not supported. No concept of state.
19 (can-a)	+	Supported via an error type trigger attached to the boundary of the activity to be cancelled.
20 (can-c)	+	Directly supported by including the entire process in a transaction. Triggering the cancel end event associated with the transaction will effectively terminate all activities associated with a process instance.
21 (str-l)	+	Both while and repeat loops are supported for individual activities and sub-processes.
22 (recur)	-	Not supported. No means of specifying recursive composition with a process model.
23 (t-trig)	-	Not supported. Triggers are supported through durable message events.
24 (p-trig)	+	Supported via message events.

Pattern	Score	Motivation
25 (can-r)	+/-	Partially supported by denoting the cancellation region as an activity set in a block activity and associating an error event with the block activity to enable cancellation, however the cancellation region is restricted to a connected subgraph of the overall process model.
26 (can-mi)	+	Achievable via a MI task which has an error type intermediate event trigger at the boundary. When the MI activity is to be withdrawn, a cancel event is triggered to terminate any remaining MI activities.
27 (comp-mi)	-	Not supported. No means of cancelling remaining MI activity instances.
28 (b-disc)	+/-	Although the COMPLEX-join gateway appears to offer support for this pattern, it is unclear how the IncomingCondition expression is specified.
29 (c-disc)	+	Supported by including the incoming branches and the OR-join in a subprocess that passes control to the following activity once the first branch has completed as well as cancelling the remaining activities in the subprocess using an error event.
30 (s-pjoin)	+/-	Although the COMPLEX-join gateway appears to offer support for this pattern, it is unclear how the IncomingCondition expression is specified.
31 (b-pjoin)	+/-	Although the COMPLEX-join gateway appears to offer support for this pattern, it is unclear how the IncomingCondition expression is specified.
32 (c-pjoin)	+/-	Although the COMPLEX-join gateway appears to offer support for this pattern, it is unclear how the IncomingCondition expression is specified.
33 (g-and-join)	+	Supported by the AND-join construct.
34 (st-pjoin-mi)	+/-	Notionally supported via the multi-instance construct where the MI Flow.Condition attribute is set to complex and ComplexMI.FlowCondition is an expression that evaluates to true when exactly M instances have completed and passes on a single token to the following activity. However it is unclear how the ComplexMI.FlowCondition is actually specified.
35 (c-pjoin-mi)	+/-	Notionally Achievable via a MI task which has an error type intermediate event trigger at the boundary. Immediately after the MI activity is an activity that issues a cancel event to terminate any remaining MI activities. However as for WCP-34, it is unclear how the ComplexMI.FlowCondition is actually specified.
36 (dyn-pjoin-mi)	-	Not supported. There is no means of adding further instances to a multi-instance task once started.

Pattern	Score	Motivation
37 (a-sync-m)	–	The OR-join gateway assumes it will be used in a structured context.
38 (g-sync-m)	–	Not supported. No means of assessing whether an OR-join gateway should fire based on a complete state analysis of the process instance.
39 (crit-sec)	–	Not supported. No means of limiting concurrent execution of a set of activities.
40 (int-rout)	+/-	Indirectly supported via an AdHoc process containing all of the activities to be interleaved with AdHocOrdering set to sequential however it is unclear what the required AdHocCompletionCondition should be to ensure each activity is executed precisely once.
41 (tm)	+	Directly supported by setting the StartQuantity attribute on activities immediately following the MI activity.
42 (ts)	+	Directly supported by setting the Quantity attribute on the outgoing sequence flow from an activity.
43 (exp-t)	+	Supported via a terminate end event.



## M UML 2.0 Activity Diagrams

This evaluation is based on UML 2.0 Activity Diagrams.

Pattern	Score	Motivation
1 (seq)	+	Directly supported by directed arcs between nodes.
2 (par-spl)	+	Supported by the ForkNode construct. It may also be represented implicitly by joining an action or activity to several subsequent actions or activities.
3 (synch)	+	Supported by the JoinNode construct. It can also be be implicitly represented.
4 (ex-ch)	+	Supported by the DecisionNode construct where the guard conditions on the outgoing edges are exclusive.
5 (simple-m)	+	Supported by the MergeNode construct.
6 (m-choice)	+	Supported by the ForkNode construct with guard conditions on the outgoing edges.
7 (s-sync-m)	–	Not supported. The specific configuration of the JoinSpec condition to achieve this is unclear.
8 (multi-m)	+	Supported by the MergeNode construct.
9 (s-disc)	+/–	The specific configuration of the JoinSpec condition to achieve this is unclear.
10 (arb-c)	+	Unstructured cycles can be captured in UML 2.0 ADs.
11 (impl-t)	+	Supported via the FlowFinalNode construct.
12 (mi-no-s)	+	Supported by spawning off new activity instances in a loop.
13 (mi-dt)	+	Supported through the use of the ExpansionRegion construct.
14 (mi-rt)	+	Supported through the use of the ExpansionRegion construct.
15 (mi-no)	–	Not supported. No means of adding additional activity instances after commencement.
16 (def-c)	+	Supported through a ForkNode and a set of AcceptSignal actions, one preceding each action in the choice.
17 (int-par)	–	Not supported. No notion of state within UML 2.0 ADs.
18 (milest)	–	Not supported. No notion of state within UML 2.0 ADs.
19 (can-a)	+	Supported by incorporating the activity in an interruptible region triggered either by a signal or execution of another activity.
20 (can-c)	+	Supported by incorporating all of the activities in the process in an interruptible region triggered either by a signal or execution of another activity.
21 (str-l)	+	Supported via the LoopNode construct.
22 (recur)	–	Not supported. No means of specifying recursive composition with a process model.

Pattern	Score	Motivation
23 (t-trig)	+	Supported using the AcceptEventAction construct for signal processing with explicit enablement.
24 (p-trig)	+	Supported via signals.
25 (can-r)	+	Supported by the InterruptibleActivityRegion construct.
26 (can-mi)	+	Supported by including the activity in an interruptible region.
27 (comp-mi)	–	No means of cancelling remaining activity instances once a multiple instance activity has commenced.
28 (b-disc)	+/-	The specific configuration of the JoinSpec condition to achieve this is unclear.
29 (c-disc)	+	Supported by incorporating the incoming branches to the join in an interruptible region. The join has an outgoing weight of 1 from the interruptible region to the subsequent activity effectively cancelling all other branches when the first branch reaches the join.
30 (s-pjoin)	+/-	The specific configuration of the JoinSpec condition to achieve this is unclear.
31 (b-pjoin)	+/-	The specific configuration of the JoinSpec condition to achieve this is unclear.
32 (c-pjoin)	+	Supported by incorporating the incoming branches to the join in an interruptible region. The join has an outgoing weight of N from the interruptible region to the subsequent activity effectively cancelling all other branches when the Nth branch reaches the join.
33 (g-and-join)	–	Not supported. JoinNode semantics prevent this situation from arising.
34 (st-pjoin-mi)	–	Not supported. A MI activity can only complete when all N instances specified in the ExpansionRegion have completed.
35 (c-pjoin-mi)	–	Not supported. A MI activity can only complete when all N instances specified in the ExpansionRegion have completed.
36 (dyn-pjoin-mi)	–	Not supported. A MI activity can only complete when all N instances specified in the ExpansionRegion have completed.
37 (a-sync-m)	+/-	The specific configuration of the JoinSpec condition to achieve this is unclear.
38 (g-sync-m)	–	Not supported. No means of determining when a join should fire based on evaluation of the overall state of the process instance.
39 (crit-sec)	–	Not supported. No means of preventing concurrent execution of a set of activities.
40 (int-rout)	–	Not supported. No notion of state within UML 2.0 ADs.

<b>Pattern</b>	<b>Score</b>	<b>Motivation</b>
41 (tm)	+	Supported by including a weighted edge after the MI activity to any subsequent activity.
42 (ts)	+	Supported by linking multiple outgoing arcs from an activity (one for each execution thread required) to a MergeNode which then directs the various initiated threads of control down the same branch.
43 (exp-t)	+	Supported via the ActivityFinalNode construct.

## N EPCs

In this evaluation we use the semantics suggested by the ARIS Toolset 6.2 (e.g., the simulator).

Pattern	Score	Motivation
1 (seq)	+	Directly supported, i.e., events can be used to connect functions in a sequence.
2 (par-spl)	+	Supported by the AND-split connector.
3 (synch)	+	Supported by the AND-join connector.
4 (ex-ch)	+	Supported by the OR-split connector. However, no language is given to describe the conditions for taking a branch.
5 (simple-m)	+	Supported by the XOR-join connector.
6 (m-choice)	+	Supported by the OR-split connector.
7 (s-sync-m)	+	Supported by the OR-join connector. However, as described in [Kin06], the OR-join connector creates a paradox when combined with loops (i.e. the “vicious circle”). The desirable semantics would be to wait until no new folder can reach the OR-join. However, one connector may depend on the other and vice versa. The ARIS simulator uses a rather odd solution for this dilemma: the OR-join connector has a time-out mechanism attached to it. The join waits for a pre-specified time and then consumes all folders that are there.
8 (multi-m)	–	Not supported. When using an XOR-join connector only one incoming folder is expected. Most papers and books on EPCs state that the XOR-join should block if multiple folders arrive. However, the ARIS simulator implements this in a slightly different way. If multiple folder arrive at exactly the same time, they are ignored. Otherwise, each folder is passed on.
9 (s-disc)	–	Not supported as there is no notion of ignoring subsequent folders and then resetting.
10 (arb-c)	+	Directly supported as there are no limitations on the graph structure.
11 (impl-t)	+	Directly supported. Note that there may be many end events and only if no element is enabled for a given instance does the instance terminate.
12 (mi-no-s)	–	This is not supported. There is no explicit notion of instances and there is no notation for inter-process connections.
13 (mi-dt)	–	Not supported. The only way to realize this is by making copies and putting them in parallel.
14 (mi-rt)	–	Not supported.
15 (mi-no)	–	Not supported.

Pattern	Score	Motivation
16 (def-c)	–	Not supported. The split and join connectors are executed when they become enabled. Therefore, an XOR-split makes an immediate decision and this choice cannot be deferred.
17 (int-par)	–	Not supported.
18 (milest)	–	Not supported. Although EPCs contain events, it is not possible to check states and act accordingly.
19 (can-a)	–	Not supported. There is no cancellation feature.
20 (can-c)	–	Not supported.
21 (str-l)	–	Not supported. There are only arbitrary cycles.
22 (recur)	–	Not supported.
23 (t-trig)	–	Not supported.
24 (p-trig)	+/-	EPCs allow for multiple input events and these could be interpreted as persistent triggers. However, since it is not possible to differentiate between the creation of an instances and subsequent events and there is no implementation of this concept, we rate this as partial support.
25 (can-r)	–	Not supported. There is no cancellation feature.
26 (can-mi)	–	Not supported.
27 (comp-mi)	–	Not supported.
28 (b-disc)	–	Not supported.
29 (c-disc)	–	Not supported.
30 (s-pjoin)	–	Not supported.
31 (b-pjoin)	–	Not supported.
32 (c-pjoin)	–	Not supported.
33 (g-and-join)	+/-	Most papers and books on EPCs assume that joins block if multiple folders arrive. However, the ARIS simulator implements this more like in a Petri net.
34 (st-pjoin-mi)	–	Not supported.
35 (c-pjoin-mi)	–	Not supported.
36 (dyn-pjoin-mi)	–	Not supported.
37 (a-sync-m)	+	See pattern 7. The language allows for this. However, we know of no workflow management systems that implement this. Note that the ARIS simulator uses a rather odd solution: the OR-join connector has a time-out mechanism attached to OR-joins, i.e., the join waits for a pre-specified time and then consumes all folders that are pending.
38 (g-sync-m)	–	See pattern 7. The language allows for this. However, we know of no workflow management systems that implement this. Note that the ARIS simulator uses a rather odd solution: the OR-join connector has a time-out mechanism attached to OR-joins, i.e., the join waits for a pre-specified time and then consumes all folders that are pending.

<b>Pattern</b>	<b>Score</b>	<b>Motivation</b>
39 (crit-sec)	–	Not supported. There is no way to create a kind of mutual exclusion state.
40 (int-rout)	–	Not supported.
41 (tm)	–	Not supported. It is impossible to merge a specified number of threads into one.
42 (ts)	–	Not supported. It is impossible to merge a specified number of threads into one.
43 (exp-t)	–	Process instances complete only when no remaining element are enabled.