Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages

W.M.P. van der Aalst[1,2] and A.H.M. ter Hofstede[2]

[1] Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tm.tue.nl
[2] Queensland University of Technology, School of Information Systems
P.O. Box 2434, Brisbane Qld 4001, Australia.
a.terhofstede@qut.edu.au

Abstract. Contemporary workflow management systems are driven by explicit process models, i.e., a completely specified workflow design is required in order to enact a given workflow process. Creating a workflow design is a complicated time-consuming process which is often hampered by the limitations of the workflow language being used. To identify the differences between the various languages, we have collected a fairly complete set of workflow patterns. Based on these patterns we have evaluated 15 workflow products and detected considerable differences in expressive power. Languages based on Petri nets perform better when it comes to state-based workflow patterns. However, some patterns (e.g. involving multiple instances, complex synchronizations or non-local withdrawals) are not easy to map onto (high-level) Petri nets. These patterns pose interesting modeling problems and are used for developing the Petri-net-based language YAWL (Yet Another Workflow Language).

1 Introduction

Workflow technology continues to be subjected to on-going development in its traditional application areas of business process modeling and business process coordination, and now in emergent areas of component frameworks and inter-workflow, business-to-business interaction. Addressing this broad and rather ambitious reach, a large number of workflow products, mainly workflow management systems (WFMS), are commercially available, which see a large variety of languages and concepts based on different paradigms (see e.g. [1, 4, 12, 19, 23, 28, 31, 30, 40, 47]).

As current provisions are compared and as newer concepts and languages are embarked upon, it is striking how little, other than standards glossaries, is available for central reference. One of the reasons attributed to the lack of consensus of what constitutes a workflow specification is the variety of ways in which business processes are otherwise described. The absence of a universal organizational "theory", and standard business process modeling concepts, it is contended, explains and ultimately justifies the major differences in workflow languages - fostering up a "horses for courses" diversity in workflow languages. What is more, the comparison of different workflow products winds up being more of a dissemination of products and less of a critique of

workflow language capabilities - "bigger picture" differences of workflow specifications are highlighted, as are technology, typically platform dependent, issues.

Workflow specifications can be understood, in a broad sense, from a number of different perspectives (see [4, 23]). The *control-flow* perspective (or process) perspective describes activities and their execution ordering through different constructors, which permit flow of execution control, e.g., sequence, choice, parallelism and join synchronization. Activities in elementary form are atomic units of work, and in compound form modularize an execution order of a set of activities. The *data perspective* layers business and processing data on the control perspective. Business documents and other objects which flow between activities, and local variables of the workflow, qualify in effect pre- and post-conditions of activity execution. The *resource perspective* provides an organizational structure anchor to the workflow in the form of human and device roles responsible for executing activities. The *operational* perspective describes the elementary actions executed by activities, where the actions map into underlying applications. Typically, (references to) business and workflow data are passed into and out of applications through activity-to-application interfaces, allowing manipulation of the data within applications.

Clearly, the control flow perspective provides an essential insight into a workflow specification's effectiveness. The data flow perspective rests on it, while the organizational and operational perspectives are ancillary. If workflow specifications are to be extended to meet newer processing requirements, control flow constructors require a fundamental insight and analysis. Currently, most workflow languages support the basic constructs of sequence, iteration, splits (AND and XOR) and joins (AND and XOR) - see [4, 30]. However, the interpretation of even these basic constructs is not uniform and it is often unclear how more complex requirements could be supported. Indeed, vendors are afforded the opportunity to recommend implementation level "hacks" such as database triggers and application event handling. The result is that neither the current capabilities of workflow languages nor insight into more complex requirements of business processes is advanced.

We indicate requirements for workflow languages through workflow *patterns* [5–8, 48]. As described in [36], a pattern "is the abstraction from a concrete form which keeps recurring in specific nonarbitrary contexts". Gamma et al. [17] first catalogued systematically some 23 design patterns which describe the smallest recurring interactions in object-oriented systems. The design patterns, as such, provided independence from the implementation technology and at the same time independence from the essential requirements of the domain that they were attempting to address (see also e.g. [15]).

We have collected a set of about 30 workflow patterns and have used 20 of these patterns to compare the functionality of 15 workflow management systems (COSA, Visual Workflow, Forté Conductor, Lotus Domino Workflow, Meteor, Mobile, MQSeries/Workflow, Staffware, Verve Workflow, I-Flow, InConcert, Changengine, SAP R/3 Workflow, Eastman, and FLOWer). The result of this evaluation reveals that (1) the expressive power of contemporary systems leaves much to be desired and (2) the systems support different patterns. Note that we do not use the term "expressiveness" in the traditional or formal sense. If one abstracts from capacity constraints, any workflow language is Turing complete. Therefore, it makes to sense to compare these languages using for-

mal notions of expressiveness. Instead we use a more intuitive notion of expressiveness which takes the modeling effort into account. This more intuitive notion is often referred to as suitability. See [27] for a discussion on the distinction between formal expressiveness and suitability.

The observation that the expressive power of the available workflow management systems leaves much to be desired, triggered the question: *How about high-level Petri nets (i.e., Petri nets extended with color, time, and hierarchy) as a workflow language?*

Petri nets have been around since the sixties [35] and have been extended with color [24, 25] and time [32, 33] to improve expressiveness. High-level Petri nets tools such as Design/CPN (University of Aarhus, http://www.daimi.au.dk/designCPN/) and ExSpect (EUT/D&T Bakkenist, http://www.exspect.com/) incorporate these extensions and support the modeling and analysis of complex systems. There are at least three good reasons for using Petri nets as a workflow language [1]:

1. Formal semantics despite the graphical nature.
2. State-based instead of (just) event-based.
3. Abundance of analysis techniques.

Unfortunately, a straightforward application of high-level Petri nets does not yield the desired result. There seem to be three problems relevant for modeling workflow processes:

1. In a high-level Petri net it is possible to use colored tokens. Although it is possible to use this to identify multiple instances of a subprocess, there is no specific support for *patterns involving multiple instances* and the burden of keeping track, splitting, and joining is carried by the designer.
2. Sometimes two flows need to be joined while it is not clear whether synchronization is needed, i.e., if both flows are active an AND-join is needed otherwise an XOR-join. Such *advanced synchronization patterns* are difficult to model in terms of a high-level Petri net because the local transition rule is either an AND-join or an XOR-join.
3. The firing of a transition is always local only based on the tokens in the input places and only affecting the output places. However, some events in the workflow may have an effect which is not local, e.g., because of an error tokens need to be removed from various places without knowing where the tokens reside. Everyone who has modeled such a *cancellation pattern* (e.g., a global timeout mechanism) in terms of Petri nets knows that it is cumbersome to model a so-called "vacuum cleaner" removing tokens from selected parts of the net.

In this paper, we discuss the problems when supporting the workflow patterns with high-level Petri nets. We also briefly introduce a workflow language under development: *YAWL (Yet Another Workflow Language)*. YAWL is based on Petri nets but extended with additional features to facilitate the modeling of complex workflows.

2 Workflow patterns

Since 1999 we have been working on collecting a comprehensive set of workflow patterns [5–8]. The results have been made available through the "Workflow patterns

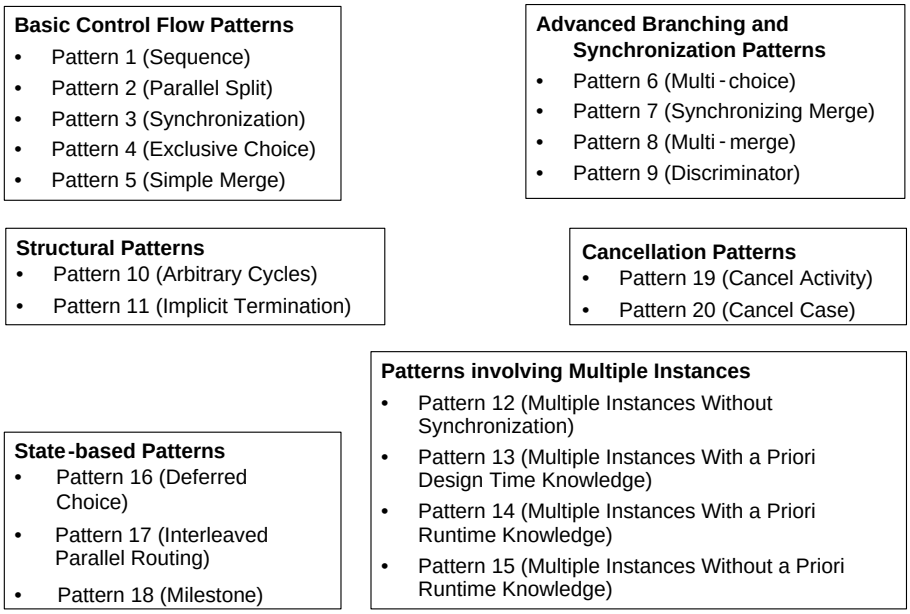


Fig. 1. Overview of the 20 most relevant patterns.

WWW site" [48]. The patterns range from very simple patterns such as sequential routing (Pattern 1) to complex patterns involving complex synchronizations such as the discriminator pattern (Pattern 9). In this paper, we restrict ourselves to the 20 most relevant patterns. These patterns can be classified into six categories:

1. *Basic control flow patterns.* These are the basic constructs present in most workflow languages to model sequential, parallel and conditional routing.
2. *Advanced branching and synchronization patterns.* These patterns transcend the basic patterns to allow for more advanced types of splitting and joining behavior. An example is the Synchronizing merge (Pattern 7) which behaves like an AND-join or XOR-join depending on the context.
3. *Structural patterns.* In programming languages a block structure which clearly identifies entry and exit points is quite natural. In graphical languages allowing for parallelism such a requirement is often considered to be too restrictive. Therefore, we have identified patterns that allow for a less rigid structure.
4. *Patterns involving multiple instances.* Within the context of a single case (i.e., workflow instance) sometimes parts of the process need to be instantiated multiple times, e.g., within the context of an insurance claim, multiple witness statements need to be processed.
5. *State-based patterns.* Typical workflow systems focus only on activities and events and not on states. This limits the expressiveness of the workflow language because it is not possible to have state dependent patterns such as the Milestone pattern (Pattern 18).

6. *Cancellation patterns.* The occurrence of an event (e.g., a customer canceling an order) may lead to the cancellation of activities. In some scenarios such events can even cause the withdrawal of the whole case.

Figure 1 shows an overview of the 20 patterns grouped into the six categories. A detailed discussion of these patterns is outside the scope of this paper. The interested reader is referred to [5–8, 48].

We have used these patterns to evaluate 15 workflow systems: COSA (Ley GmbH, [43]) , Visual Workflow (Filenet, [13]), Forté Conductor (SUN, [14]), Lotus Domino Workflow (IBM/Lotus, [34]), Meteor (UGA/LSDIS, [41]), Mobile (UEN, [23]), MQ-Series/Workflow (IBM, [22]), Staffware (Staffware PLC, [44]), Verve Workflow (Versata, [46]), I-Flow (Fujitsu, [16]), InConcert (TIBCO, [45]), Changengine (HP, [21]), SAP R/3 Workflow (SAP, [39]), Eastman (Eastman, [42]), and FLOWer (Pallas Athena, [9]). Tables 1 and 2 summarize the results of the comparison of the workflow management systems in terms of the selected patterns. For each product-pattern combination, we checked whether it is possible to realize the workflow pattern with the tool. If a product directly supports the pattern through one of its constructs, it is rated +. If the pattern is not *directly* supported, it is rated +/-. Any solution which results in spaghetti diagrams or coding, is considered as giving no direct support and is rated -.

pattern	product							
	Staffware	COSA	InConcert	Eastman	FLOWer	Domino	Meteor	Mobile
1 (seq)	+	+	+	+	+	+	+	+
2 (par-spl)	+	+	+	+	+	+	+	+
3 (synch)	+	+	+	+	+	+	+	+
4 (ex-ch)	+	+	+/-	+	+	+	+	+
5 (simple-m)	+	+	+/-	+	+	+	+	+
6 (m-choice)	-	+	+/-	+/-	-	+	+	+
7 (sync-m)	-	+/-	+	+	-	+	-	-
8 (multi-m)	-	-	-	+	+/-	+/-	+	-
9 (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)	-	-	-	-	+/-	+	-	-

Table 1. The main results for Staffware, COSA, InConcert, Eastman, FLOWer, Lotus Domino Workflow, Meteor, and Mobile.

pattern	product						
	MQSeries	Forté	Verve	Vis. WF	Changeng.	I-Flow	SAP/R3
1 (seq)	+	+	+	+	+	+	+
2 (par-spl)	+	+	+	+	+	+	+
3 (synch)	+	+	+	+	+	+	+
4 (ex-ch)	+	+	+	+	+	+	+
5 (simple-m)	+	+	+	+	+	+	+
6 (m-choice)	+	+	+	+	+	+	+
7 (sync-m)	+	-	-	-	-	-	-
8 (multi-m)	-	+	+	-	-	-	-
9 (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)	-	+	+	-	+	-	+

Table 2. The main results for MQSeries, Forté Conductor, Verve, Visual WorkFlo, Changengine, I-Flow, and SAP/R3 Workflow.

Please apply the results summarized in tables 1 and 2 with care. First of all, the organization selecting a workflow management system should focus on the patterns most relevant for the workflow processes at hand. Since support for the more advanced patterns is limited, one should focus on the patterns most needed. Second, the fact that a pattern is not directly supported by a product does not imply that it is not possible to support the construct at all.

From the comparison it is clear that no tool supports all the of the 20 selected patterns. In fact, many of the tools only support a relatively small subset of the more advanced patterns (i.e., patterns 6 to 20). Specifically the limited support for the discriminator, and its generalization, the N-out-of-M-join, the state-based patterns (only COSA), the synchronization of multiple instances (only FLOWer) and cancellation activities/cases, is worth noting.

The goal of providing the two tables is not to advocate the use of specific tools. However, they illustrate that existing tools and languages are truly different and that most languages provide only partial support for the patterns appearing in real life workflow processes. These observations have been our main motivation to look into the expressiveness of high-level Petri nets (Section 3) and come up with a new language (Section 4).

3 Limitations of Petri nets

Given the fact that workflow management systems have problems dealing with workflow patterns it is interesting to see whether established process modeling techniques such as Petri nets can cope with these patterns. The table listed in the appendix shows an evaluation of high-level Petri nets with respect to the patterns. (Ignore the column under YAWL for the time being.) We use the term high-level Petri nets to refer to Petri nets extended with color (i.e., data), time, and hierarchy [4]. Examples of such languages are the colored Petri nets as described in [25], the combination of Petri nets and Z specification described in [20], and many more. These languages are used by tools such as Design/CPN (University of Aarhus, http://www.daimi.au.dk/designCPN/) and ExSpect (EUT/D&T Bakkenist, http://www.exspect.com/). Although these languages and tools have differences when it comes to for example the language for data transformations (e.g., arc inscriptions) there is a clear common denominator. When we refer to high-level Petri nets we refer to this common denominator. To avoid confusion we use the terminology as defined in [25] as much as possible. It is important to note that for the table shown in the appendix we have used the same criteria as used in tables 1 and 2 for the 15 workflow systems (i.e., a "+" is only given if there is direct support).

Compared to existing languages high-level Petri nets are quite expressive. Recall that we use the term "expressiveness" not in the formal sense. High-level Petri nets are Turing complete, and therefore, can do anything we can define in terms of an algorithm. However, this does not imply that the modeling effort is acceptable. By comparing the table in the appendix with tables 1 and 2, we can see that high-level nets, in contrast to many workflow languages, have no problems dealing with state-based patterns. This is a direct consequence of the fact that Petri nets use places to represent states explicitly. Although high-level Petri nets outperform most of the existing languages, the result is not completely satisfactory. As indicated in the introduction we see serious limitations when it comes to (1) patterns involving multiple instances, (2) advanced synchronization patterns, and (3) cancellation patterns. In the remainder of this section we discuss these limitations in more detail.

3.1 Patterns involving multiple instances

Suppose that in the context of a workflow for processing insurance claims there is a subprocess for processing witness statements. Each insurance claim may involve zero or more witness statements. Clearly the number of witness statements is not known at design time. In fact, while a witness statement is being processed other witnesses may pop up. This means that within one case a part of the process needs to be instantiated a variable number of times and the number of instances required is only known at run time. The required pattern to model this situation is Pattern 15 (Multiple instances without a priori runtime knowledge). Another example of this pattern is the process of handling journal submissions. For processing journal submissions multiple reviews are needed. The editor of the journal may decide to ask a variable number of reviewers depending on the nature of the paper, e.g., if it is controversial, more reviewers are selected. While the reviewing takes place, the editor may decide to involve more reviewers. For example, if reviewers are not responsive, have brief or conflicting reviews,

then the editor may add an additional reviewer. Other examples of multiple instances include orders involving multiple items (e.g., a customer orders three books from an electronic bookstore), a subcontracting process with multiple quotations, etc.

It is possible to model a variable number of instances executed in parallel using a high-level Petri net. However, the designer of such a model has to keep track of two things: (1) case identities and (2) the number of instances still running.

At the same time multiple cases are being processed. Suppose x and y are two active cases. Whenever, there is an AND-join only tokens referring to the same case can be synchronized. If inside x part of the process is instantiated n times, then there are n "child cases" $x.1 \ldots x.n$. If for y the same part is also instantiated multiple times, say m, then there are m "child cases" $y.1 \ldots y.m$. Inside the part which is instantiated multiple times there may again be parallelism and there may be multiple tokens referring to one child case. For a normal AND-join only tokens referring to the same child case can be synchronized. However, at the end of the part which is instantiated multiple times all child cases having the same parent should be synchronized, i.e., case x can only continue if for each child case $x.1 \ldots x.n$ the part has been processed. In this synchronization child cases $x.1 \ldots x.n$ and child cases $y.1 \ldots y.m$ should be clearly separated. To complicate matters the construct of multiple instances may be nested resulting in child-child cases such as x.5.3 which should be synchronized in the right way. Clearly, a good workflow language does not put the burden of keeping track of these instances and synchronizing them at the right level on the workflow designer.

Besides keeping track of identities and synchronizing them at the right level, it is important to know how many child cases need to be synchronized. This is of particular relevance if the number of instances can change while the instances are being processed (e.g., a witness which points out another witness causing an additional witness statement). In a high-level Petri net this can be handled by introducing a counter keeping track of the number of active instances. If there are no active instances left, the child cases can be synchronized. Clearly, it is also not acceptable to put the burden of modeling such a counter on the workflow designer.

3.2 Advanced synchronization patterns

Consider the workflow process of booking a business trip. A business trip may involve the booking of flights, the booking of hotels, the booking of a rental car, etc. Suppose that the booking of flights, hotels, and cars can occur in parallel and that each of these elements is optional. This means that one trip may involve only a flight, another trip may involve a flight and a rental car, and it is even possible to have a hotel and a rental car (i.e., no flight). The process of booking each of these elements has a separate description which may be rather complex. Somewhere in the process these optional flows need to be synchronized, e.g., activities related to payment are only executed after all booking elements (i.e., flight, hotel, and car) have been processed. The problem is that it is not clear which subflows need to be synchronized. For a trip not involving a flight, one should not wait for the completion of booking the flight. However, for a business trip involving all three elements, all flows should be synchronized. The situation where there is sometimes no synchronization (XOR-join), sometimes full synchronization (AND-

join), and sometimes only partial synchronization (OR-join) needed is referred to as Pattern 7 (Synchronizing merge).

It is interesting to note that the Synchronizing merge is directly supported by InConcert, Eastman, Domino Workflow, and MQSeries Workflow. In each of these systems, the designer does not have to specify the type of join; this is automatically handled by the system.

In a high-level Petri net each construct is either an AND-join (transition) or an XOR-join (place). Nevertheless, it is possible to model the Synchronizing merge in various ways. First of all, it is possible to pass information from the split node to the join node. For example, if the business trip involves a flight and a hotel, the join node is informed that it should only synchronize the flows corresponding to these two elements. This can be done by putting a token in the input place of the synchronization transition corresponding to the element car rental. Second, it is possible to activate each branch using a "Boolean" token. If the value of the token is true, everything along the branch is executed. If the value is false, the token is passed through the branch but all activities on it are skipped. Third, it is possible to build a completely new scheduler in terms of high-level Petri nets. This scheduler interprets workflow processes and uses the following synchronization rule: "Fire a transition t if at least one of the input places of t is marked and from the current marking it is not possible to put more tokens on any of the other input places of t." In this last solution, the problem is lifted to another level. Clearly, none of the three solutions is satisfactory. The workflow designer has to add additional logic to the workflow design (case 1), has to extend the model to accommodate true and false tokens (case 2), or has to model a scheduler and lift the model to another level (case 3).

It is interesting to see how the problem of the Synchronizing merge has been handled in existing systems and literature. In the context of MQSeries workflow the technique of "dead-path elimination" is used [31, 22]. This means that initially each input arc is in state "unevaluated". As long as one of the input arcs is in this state, the activity is not enabled. The state of an input arc is changed to true the moment the preceding activity is executed. However, to avoid deadlocks the input arc is set to false the moment it becomes clear that it will not fire. By propagating these false signals, no deadlock is possible and the resulting semantics matches Pattern 7. The solution used in MQSeries workflow is similar to having true and false tokens (case 2 described above). The idea of having true and false tokens to address complex synchronizations was already raised in [18]. However, the bipolar synchronization schemes presented in [18] are primarily aimed at avoiding constructs such as the Synchronizing merge, i.e., the nodes are pure AND/XOR-splits/joins and partial synchronization is not supported nor investigated. In the context of Event-driven Process Chains (EPC's, cf. [26]) the problem of dealing with the Synchronizing merge also pops up. The EPC model allows for so-called $\vee$-connectors (i.e., OR-joins which only synchronize the flows that are active). The semantics of these $\vee$-connectors have been often debated [3, 11, 29, 37, 38]. In [3] the explicit modeling is advocated (case 1). Dehnert and Rittgen [11] advocate the use of a weak correctness notion (relaxed soundness) and an intelligent scheduler (case 3). Langner et al. [29] propose an approach based on Boolean tokens (case 2). Rump [38] proposes an intelligent scheduler to decide whether an $\vee$-connector should synchronize

or not (case 3). In [37] three different join semantics are proposed for the $\vee$-connector: (1) *wait for all to come* (corresponds to the Synchronizing merge, Pattern 7), (2) *wait for first to come and ignore others* (corresponds to the Discriminator, Pattern 9), and (3) *never wait, execute every time* (corresponds to the Multi merge, Pattern 8). The extensive literature on the synchronization problems in EPC's and workflow systems illustrates that patterns like the Synchronizing merge are relevant and far from trivial.

3.3 Cancellation patterns

Most workflow modeling languages, including high-level nets, have local rules directly relating the input of an activity to output. For most situations such local rules suffice. However, for some events local rules can be quite problematic. Consider for example the processing of Customs declarations. While a Customs declaration is being processed, the person who filed the declaration can still supply additional information and notify Customs of changes (e.g., a container was wrecked, and therefore, there will be less cargo as indicated on the first declaration). These changes may lead to the withdrawal of a case from specific parts of the process or even the whole process. Such cancellations are not as simple as they seem when for example high-level Petri nets are used. The reason is that the change or additional declaration can come at any time (within a given time frame) and may affect running and/or scheduled activities. Given the local nature of Petri net transitions, such changes are difficult to handle. If it is not known where in the process the tokens reside when the change or additional declaration is received, it is not trivial to remove these tokens. Inhibitor arcs allow for testing whether a place contains a token. However, quite some bookkeeping is required to remove tokens from an arbitrary set of places. Consider for example 10 parallel branches with 10 places each. To remove 10 tokens (one in each parallel branch) one has to consider 10^{10} possible states. Modeling a "vacuum cleaner", i.e., a construct to remove the 10 tokens, is possible but results in a spaghetti-like diagram. Therefore it is difficult to deal with cancellation patterns such as Cancel activity (Pattern 19) and Cancel case (Pattern 20) and anything in-between.

In this section we have discussed serious limitations of high-level Petri nets when it comes to (1) patterns involving multiple instances, (2) advanced synchronization patterns, and (3) cancellation patterns. Again, we would like to stress that high-level Petri nets are able to express such routing patterns. However, the modeling effort is considerable, and although the patterns are needed frequently, the burden of keeping track of things is left to the workflow designer.

4 YAWL: Yet Another Workflow Language

In a joint effort between Eindhoven University of Technology and Queensland University of Technology we are currently working on a new workflow language based on Petri nets. The goal of this joint effort is to overcome the limitations mentioned in the previous section by adding additional constructs. A detailed description of the language is beyond the scope of this paper. Moreover, the language is still under development.

The goal of this section is to briefly sketch the features of this language named *YAWL (Yet Another Workflow Language)*.

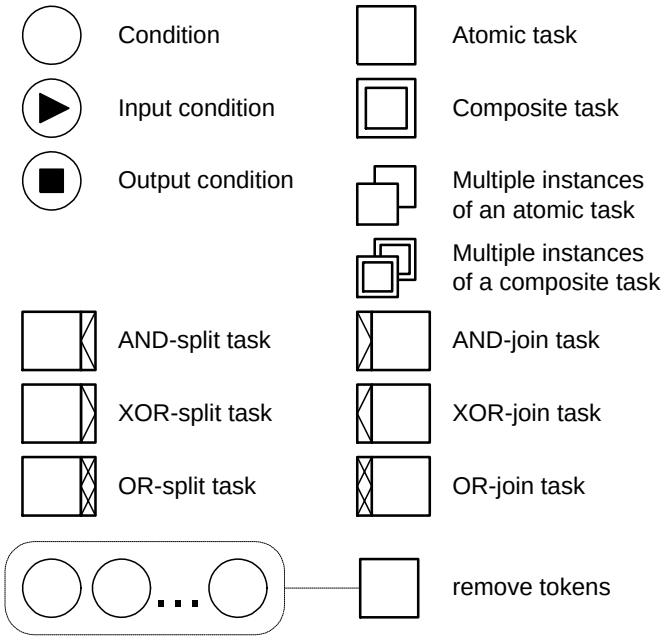


Fig. 2. Symbols used in YAWL.

Figure 2 shows the modeling elements of YAWL. YAWL extends the class of workflow nets described in [2, 4] with multiple instances, composite tasks, OR-joins, removal of tokens, and directly connected transitions. A *workflow specification* in YAWL is a set of *extended workflow nets* (EWF-nets) which form a hierarchy, i.e., there is a tree-like structure. *Tasks*[1] are either *(atomic) tasks* or *composite tasks*. Each composite task refers to a unique EWF-net at a lower level in the hierarchy. Atomic tasks form the leaves of the tree-like structure. There is one EWF-net without a composite task referring to it. This EWF-net is named the *top level workflow* and forms the root of the tree-like structure.

Each EWF-net consists of tasks (either composite or atomic) and *conditions* which can be interpreted as places. Each EWF-net has one unique *input condition* and one unique *output condition* (see Figure 2). In contrast to Petri nets, it is possible to connect "transition-like objects" like composite and atomic tasks directly to each other without using a "place-like object" (i.e., conditions) in-between. For the semantics this construct can be interpreted as a hidden condition, i.e., an implicit condition is added for every direct connection.

[1] Note that in YAWL we use the term *task* rather than *activity* to remain consistent with earlier work on workflow nets [2, 4].

Each task (either composite or atomic) can have multiple instances as indicated in Figure 2. It is possible to specify a lower bound and an upper bound for the number of instances created after initiating the task. Moreover, it is possible to indicate that the task terminates the moment a certain threshold of instances has completed. The moment this threshold is reached, all running instances are terminated and the task completes. If no threshold is specified, the task completes once all instances have completed. Finally, there is a fourth parameter indicating whether the number of instances is fixed after creating the instance. The value of the parameter is "fixed" if after creation no instances can be added and "var" if it is possible to add additional instances while there are still instances being processed. Note that by extending Petri-nets with this construct with four parameters (lower bound, upper bound, threshold, and fixed/var), we directly support all patterns involving multiple instances (cf. Section 3.1, and in addition, the Discriminator pattern (Pattern 9) under the assumption of multiple instances of the same task. In fact, we also support the more general n-out-of-m join [6].

We adopt the notation described in [2, 4] for AND/XOR-splits/joins as shown in Figure 2. Moreover, we introduce OR-splits and OR-joins corresponding to respectively Pattern 6 (Multi choice) and Pattern 7 (Synchronizing merge), cf. Section 3.2.

Finally, we introduce a notation to remove tokens from places independent of the fact if and how many tokens there are. As Figure 2 shows this is denoted by dashed circles/lines. The enabling of the task does not depend on the tokens within the dashed area. However, the moment the task executes all tokens in this area are removed. Clearly, this extension is useful for the cancellation patterns, cf. Section 3.3. Independently, this extensions was also proposed in [10] for the purpose of modeling dynamic workflows.

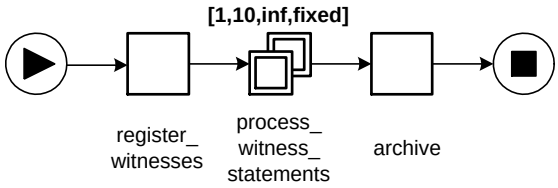
As indicated earlier, YAWL is still under development and the goal of this paper is not to introduce the language in any detail. Therefore, we restrict ourselves to simply applying YAWL to some of the examples used in the previous section.

4.1 Example: Patterns involving multiple instances

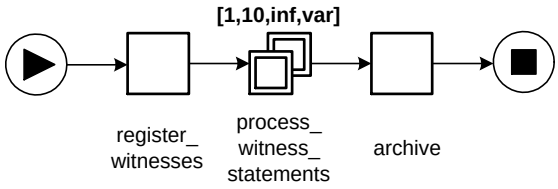
Figure 3 shows three workflow specifications dealing with multiple witness statements in parallel. The first workflow specification (a), starts between 1 and 10 instances of the composite task *process_witness_statement* after completing the initial task *register_witness*. When all instances have completed, task *archive* is executed. The second workflow specification shown in Figure 3(b), starts an arbitrary number of instances of the composite task and even allows for the creation of new instances. The third workflow specification (c) starts between 1 and 10 instances of the composite task *process_witness_statement* but the finishes if all have completed or at least three have completed. The three examples illustrate that YAWL allows for a direct specification of the patterns 14, 15, and 9.

4.2 Example: Advanced synchronization patterns

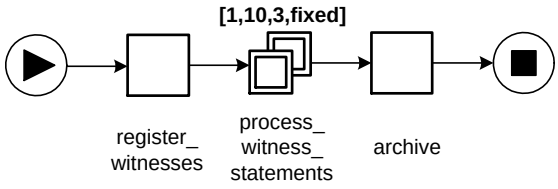
As explained in Section 3.2 an OR-join can be interpreted in many ways. Figure 4 shows three possible interpretations using the booking of a business trip as an example. The first workflow specification (a) starts with an OR-split *register* which enables tasks *flight*, *hotel* and/or *car*. Task *pay* is executed for each time one of the three tasks (i.e.,



(a) A workflow processing between 1 and 10 witness statements without the possibility to add witnesses after registration (Pattern 14).

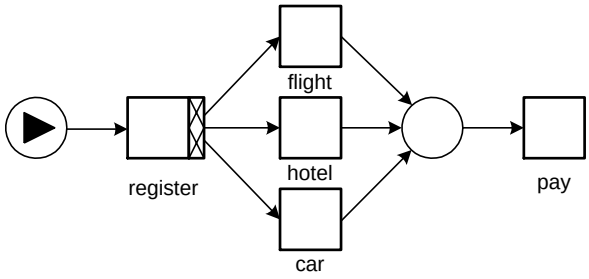


(b) A workflow processing and arbitrary number of witnesses with the possibility to add new batches of witnesses (Pattern 15).

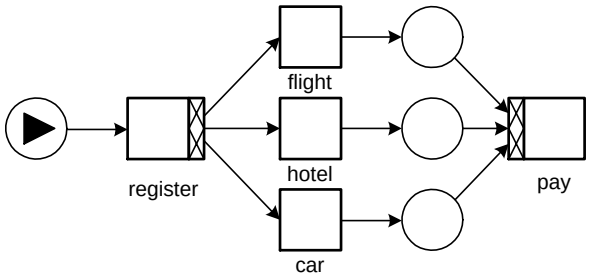


(c) A workflow processing between 1 and 10 witness statements with a threshold of 3 witnesses (extension of Pattern 9).

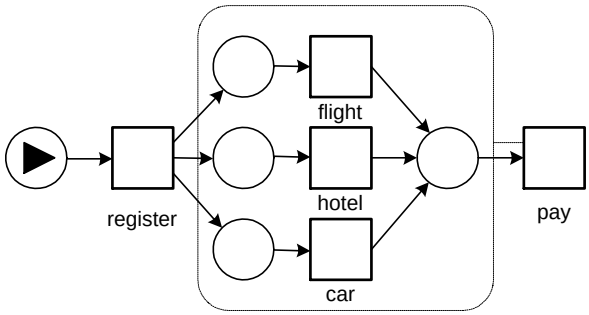
Fig. 3. Some examples illustrating the way YAWL deals with multiple instances.



(a) Task pay is executed each time one of the three preceding task completes (Pattern 8).



(b) Task pay is executed only once, i.e., when all started tasks have completed (Pattern 7).



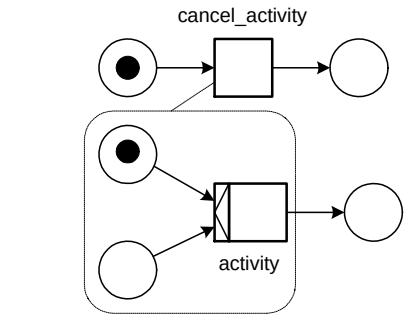
(c) Task pay is executed only once, i.e., when the first task has completed (Pattern 9).

Fig. 4. Some examples illustrating the way YAWL deals with advanced synchronization patterns.

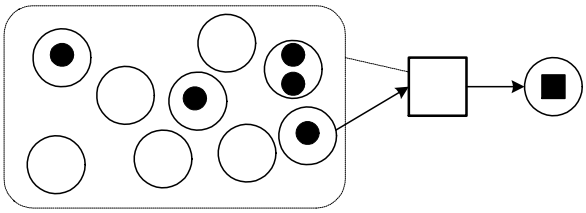
flight, *hotel*, and *car*) completes. This construct corresponds to the Multi merge (Pattern 8). The second workflow specification shown in Figure 4(b) is similar but combines the individual payments into one payment. Therefore, it waits until each of the tasks enabled by *register* completes. Note that if only a flight is booked, there is no synchronization. However, if the trip contains two or even three elements, task *pay* is delayed until all have completed. This construct corresponds to the Synchronizing merge (Pattern 7). The third workflow specification (c) enables all three tasks (i.e., *flight*, *hotel*, and *car*) but pays after the first task is completed. After the payment all running tasks are canceled. Although this construct makes no sense in this context it has been added to illustrate how the Discriminator can be supported (Pattern 9) assuming that all running threads are canceled the moment the first one completes.

4.3 Example: Cancellation patterns

Figure 5 illustrates the way YAWL supports the two cancellation patterns (patterns 19 and 20). The first workflow specification (a) shows the Cancel activity pattern which removes all tokens from the input places of task *activity*. In the second workflow speci-



(a) Cancel activity (Pattern 19).



(b) Cancel case (Pattern 20).

Fig. 5. Some examples illustrating the way YAWL deals with cancellation patterns.

fication (b) there is a task removing all tokens and putting a token in the output condition thus realizing the Cancel case pattern.

The examples given in this section illustrate that YAWL solves many of the problems indicated in Section 3. The table in the appendix shows that YAWL supports 19 of the 20 patterns used to evaluate contemporary workflow systems. Implicit termination (i.e., multiple output conditions) is not supported to force the designer to think about termination properties of the workflow. It would be fairly easy to extend YAWL with this pattern (simply connect all output conditions with an OR-join having a new and unique output condition). However, implicit termination also hides design errors because it is not possible to detect deadlocks. Therefore, there is no support for this pattern.

5 Conclusion

The workflow patterns described in previous publications [5–8, 48] provide functional requirements for workflow languages. Unfortunately, existing workflow languages only offer partial support for these patterns. Compared with the workflow languages used by commercial tools, high-level Petri nets are acceptable. Nevertheless, when it comes to patterns involving multiple instances, advanced synchronization, and cancellation, high-level Petri nets offer little support. Therefore, we are working towards a more expressive Petri-net-based language supporting most patterns. Moreover, we hope that the modeling problems collected in this paper will stimulate other researchers working on high-level Petri nets to develop mechanisms, tools, and methods providing more support.

References

1. W.M.P. van der Aalst. Chapter 10: Three Good reasons for Using a Petri-net-based Workflow Management System. In T. Wakayama, S. Kannapan, C.M. Khoong, S. Navathe, and J. Yates, editors, *Information and Process Integration in Enterprises: Rethinking Documents*, volume 428 of *The Kluwer International Series in Engineering and Computer Science*, pages 161–182. Kluwer Academic Publishers, Boston, Massachusetts, 1998.
2. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
3. W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999.
4. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, Berlin, 2000.
6. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, 2000.

7. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Technical report, Eindhoven University of Technology, Eindhoven, 2002. http://www.tm.tue.nl/it/research/patterns.
8. W.M.P. van der Aalst and H. Reijers. Adviseurs slaan bij workflow-systemen de plank regelmatig mis. *Automatisering Gids*, 36(15):15–15, 2002.
9. Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2001.
10. P. Chrzastowski-Wachtel. Top-down Petri Net Based Approach to Dynamic Workflow Modeling (Work in Progress). University of New South Wales, Sydney, 2002.
11. J. Dehnert and P. Rittgen. Relaxed Soundness of Business Processes. In K.R. Dittrich, A. Geppert, and M.C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 157–170. Springer-Verlag, Berlin, 2001.
12. C.A. Ellis and G.J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, Berlin, 1993.
13. FileNet. *Visual WorkFlo Design Guide*. FileNet Corporation, Costa Mesa, CA, USA, 1997.
14. Forté. *Forté Conductor Process Development Guide*. Forté Software, Inc, Oakland, CA, USA, 1998.
15. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, Massachusetts, 1997.
16. Fujitsu. *i-Flow Developers Guide*. Fujitsu Software Corporation, San Jose, CA, USA, 1999.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.
18. H. J. Genrich and P. S. Thiagarajan. A Theory of Bipolar Synchronization Schemes. *Theoretical Computer Science*, 30(3):241–318, 1984.
19. D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.
20. K.M. van Hee. *Information System Engineering: a Formal Approach*. Cambridge University Press, 1994.
21. HP. *HP Changengine Process Design Guide*. Hewlett-Packard Company, Palo Alto, CA, USA, 2000.
22. IBM. *IBM MQSeries Workflow - Getting Started With Buildtime*. IBM Deutschland Entwicklung GmbH, Boeblingen, Germany, 1999.
23. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
24. K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer-Verlag, Berlin, 1990.
25. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.
26. G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.
27. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002.
28. T.M. Koulopoulos. *The Workflow Imperative*. Van Nostrand Reinhold, New York, 1995.
29. P. Langner, C. Schneider, and J. Wehler. Petri Net Based Certification of Event driven Process Chains. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998*, volume 1420 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, Berlin, 1998.

30. P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997.
31. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
32. M. Ajmone Marsan, G. Balbo, and G. Conte. A Class of Generalised Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.
33. M. Ajmone Marsan, G. Balbo, and G. Conte et al. *Modelling with Generalized Stochastic Petri Nets*. Wiley series in parallel computing. Wiley, New York, 1995.
34. S.P. Nielsen, C. Easthope, P. Gosselink, K. Gutsze, and J. Roele. *Using Lotus Domino Workflow 2.0, Redbook SG24-5963-00*. IBM, Poughkeepsie, USA, 2000.
35. C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.
36. D. Riehle and H. Züllighoven. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.
37. P. Rittgen. Modified EPCs and their Formal Semantics. Technical report 99/19, University of Koblenz-Landau, Koblenz, Germany, 1999.
38. F. Rump. Erreichbarkeitsgraphbasierte Analyse ereignisgesteuerter Prozessketten. Technischer Bericht, Institut OFFIS, 04/97 (in German), University of Oldenburg, Oldenburg, 1997.
39. SAP. *WF SAP Business Workflow*. SAP AG, Walldorf, Germany, 1997.
40. T. Schäl. *Workflow Management for Process Organisations*, volume 1096 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.
41. A. Sheth, K. Kochut, and J. Miller. Large Scale Distributed Information Systems (LSDIS) laboratory, METEOR project page. http://lsdis.cs.uga.edu/proj/meteor/meteor.html.
42. Eastman Software. *RouteBuilder Tool User's Guide*. Eastman Software, Inc, Billerica, MA, USA, 1998.
43. Software-Ley. *COSA 3.0 User Manual*. Software-Ley GmbH, Pullheim, Germany, 1999.
44. Staffware. *Staffware 2000 / GWD User Manual*. Staffware plc, Berkshire, United Kingdom, 2000.
45. Tibco. *TIB/InConcert Process Designer User's Guide*. Tibco Software Inc., Palo Alto, CA, USA, 2000.
46. Verve. *Verve Component Workflow Engine Concepts*. Verve, Inc., San Francisco, CA, USA, 2000.
47. WFMC. Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition, Brussels, 1996.
48. Workflow Patterns Home Page. http://www.tm.tue.nl/it/research/patterns.

A A comparison of high-level Petri nets and YAWL using the patterns

The table shown in this appendix indicates for each pattern whether high-level Petri nets/YAWL offers direct support (indicated by a "+"), partial direct support (indicated by a "+/-"), or no direct support (indicated by a "-").

pattern	high-level Petri nets	YAWL
1 (seq)	+	+
2 (par-spl)	+	+
3 (synch)	+	+
4 (ex-ch)	+	+
5 (simple-m)	+	+
6 (m-choice)	+	+
7 (sync-m)	$-^{(i)}$	+
8 (multi-m)	+	+
9 (disc)	$-^{(ii)}$	+
10 (arb-c)	+	+
11 (impl-t)	$-^{(iii)}$	$-^{(iv)}$
12 (mi-no-s)	+	+
13 (mi-dt)	+	+
14 (mi-rt)	$-^{(v)}$	+
15 (mi-no)	$-^{(vi)}$	+
16 (def-c)	+	+
17 (int-par)	+	+
18 (milest)	+	+
19 (can-a)	$+/-^{(vii)}$	+
20 (can-c)	$-^{(viii)}$	+

(i) The synchronizing merge is not supported because the designer has to keep track of the number of parallel threads and decide to merge or synchronize flows (cf. Section 3.2).

(ii) The discriminator is not supported because the designer needs to keep track of the number of threads running and the number of threads completed and has to reset the construct explicitly by removing all tokens corresponding to the iteration (cf. Section 3.2).

(iii) Implicit termination is not supported because the designer has to keep track of running threads to decide whether the case is completed.

(iv) Implicit termination is not supported because the designer is forced to identify one unique final node. Any model with multiple end nodes can be transformed into a net with a unique end node (simply use a synchronizing merge). This has not been added to YAWL to force the designer to think about successful completion of the case. This requirement allows for the detection of unsuccessful completion (e.g., deadlocks).

(v) Multiple instances with synchronization are not supported by high-level Petri nets (cf. Section 3.1).

(vi) Also not supported, cf. Section 3.1.

(vii) Cancel activity is only partially supported since one can remove tokens from the input place of a transition but additional bookkeeping is required if there are multiple input places and these places may be empty (cf. Section 3.3).

(viii) Cancel activity is not supported because one needs to model a vacuum clearer to remove tokens which may of may not reside in specific places (cf. Section 3.3).