YAWL: Yet Another Workflow Language (Revised version)

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] Centre for Information Technology Innovation, Queensland University of Technology
P.O. Box 2434, Brisbane Qld 4001, Australia.
a.terhofstede@qut.edu.au

Abstract. Based on a rigorous analysis of existing workflow management systems and workflow languages, a new workflow language is proposed: YAWL (Yet Another Workflow Language). 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 several workflow products and detected considerable differences in their ability to capture control flows for non-trivial workflow processes. Languages based on Petri nets perform better when it comes to state-based workflow patterns. However, some patterns (e.g. involving multiple instances, complex synchronisations or non-local withdrawals) are not easy to map onto (high-level) Petri nets. This inspired us to develop a new language by taking Petri nets as a starting point and adding mechanisms to allow for a more direct and intuitive support of the workflow patterns identified. This paper motivates the need for such a language, specifies the semantics of the language, and shows that soundness can be verified in a compositional way. Although YAWL is intended as a complete workflow language, the focus of this paper is limited to the control-flow perspective.

1 Introduction

Despite the efforts of the Workflow Management Coalition (WfMC, [40, 17]), workflow management systems use a large variety of languages and concepts based on different paradigms. Most of the products available use a proprietary language rather than a tool-independent language. Some workflow management systems are based on Petri nets but typically add both product specific extensions and restrictions [1, 5, 15]. Other systems use a completely different mechanism. For example, IBM's MQSeries Workflow uses both active and passive threads rather than token passing [41]. The differences between the various tools are striking. 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 organisational "theory", and standard business process modelling 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 [6].

Workflow specifications can be understood, in a broad sense, from a number of different perspectives (see [5, 32]). The *control-flow* perspective (or process) perspective describes tasks and their execution ordering through different constructors, which permit flow of execution control, e.g., sequence, choice, parallelism and join synchronisation. Tasks in elementary form are atomic units of work, and in compound form modularise an execution order of a set of tasks. The *data perspective* deals with business and processing data. This perspective is layered on top of 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 task execution. The *resource perspective* provides an organisational structure anchor to the workflow in the form of human and device roles responsible for executing tasks. The *operational* perspective describes the elementary actions executed by tasks, 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.

The focus of this paper is on the control-flow perspective. Clearly, this provides an essential insight into a workflow specification's effectiveness. The data flow perspective rests on it, while the organisational 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 [5, 40]. 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". The result is that neither the current capabilities of workflow languages nor insight into more complex requirements of business processes is advanced [6].

We indicate requirements for workflow languages through workflow *patterns* [6, 70]. As described in [50], a pattern "is the abstraction from a concrete form which keeps recurring in specific nonarbitrary contexts". Gamma et al. [22] 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. [19]).

We have collected a comprehensive set of workflow 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 no sense to compare these languages using formal notions of expressiveness. Instead we use a more intuitive notion of expressiveness which takes the modelling effort into account. This more intuitive notion is often referred to as *suitability*. See [36] for a discussion on the distinction between formal expressiveness and suitability. In the remainder, we will use the term suitability.

The observation that the suitability of the available workflow management systems leaves much to be desired, triggered the question: *How about high-level Petri nets as a workflow language?*

Petri nets have been around since the sixties [47] and have been extended with colour [33, 34] and time [42] to improve expressiveness. Petri nets where tokens carry data (i.e., are coloured) are often referred to as high-level Petri nets and are supported by tools such as Design/CPN (University of Aarhus, http://www.daimi.au.dk/designCPN/), CPN Tools (University of Aarhus, http://www.daimi.au.dk/CPNTools/), and ExSpect (EUT/D&T Bakkenist, http://www.exspect.com/). Note that these tools and the corresponding languages also allow for time and mechanisms to hierarchically structure complex models. Therefore, we use the term *high-level Petri nets* to refer to Petri nets extended with colour, time and hierarchy.

There are at least three good reasons for using Petri net based workflow languages [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 modelling workflow processes:

1. High-level Petri nets support coloured tokens, i.e., a token can have a value. 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 of instances is carried by the designer.
2. Sometimes two flows need to be joined while it is not clear whether synchronisation is needed, i.e., if both flows are active an AND-join is needed otherwise an XOR-join. Such *advanced synchronisation patterns* are difficult to model in terms of a high-level Petri net because the firing rule only supports two types of joins: the AND-join (transition) or the XOR-join (place).
3. The firing of a transition is always local, i.e., enabling is only based on the tokens in the input places and firing is only affecting the input and 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 modelled 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. Based on this we describe *YAWL (Yet Another Workflow Language)*.

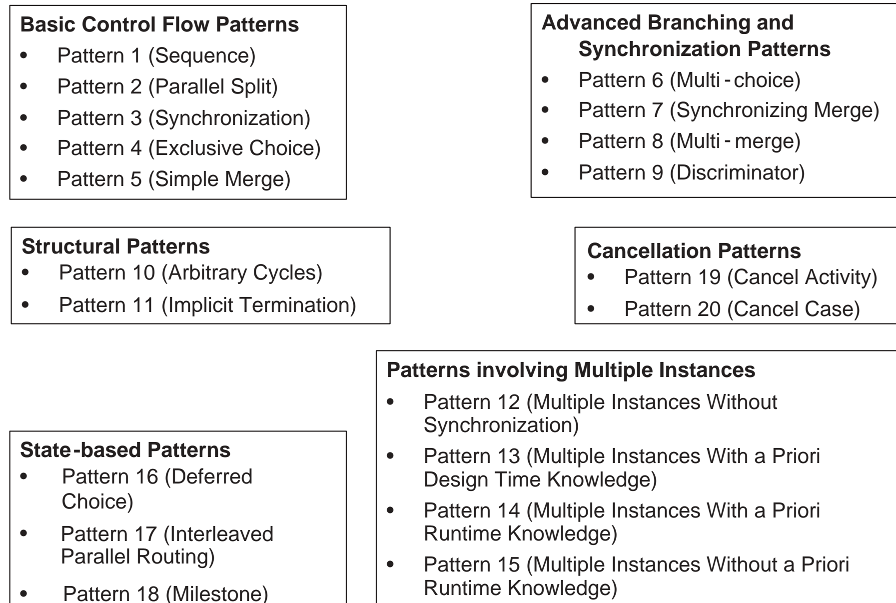


Fig. 1. Overview of the 20 workflow patterns described in [6].

YAWL is based on Petri nets but extended with additional features to facilitate the modelling of complex workflows.

2 Requirements

Since 1999 we have been working on collecting a comprehensive set of workflow patterns [6]. The results have been made available through the "Workflow patterns WWW site" [70]. The patterns range from very simple patterns such as sequential routing (Pattern 1) to complex patterns involving complex synchronisations 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 synchronisation patterns.* These patterns transcend the basic patterns to allow for more advanced types of splitting and joining behaviour. An example is the Synchronising 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 cancelling 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 [6, 70].

We have used these patterns to evaluate 15 workflow systems: COSA (Ley GmbH, [58]) , Visual Workflow (Filenet, [16]), Forté Conductor (SUN, [18]), Lotus Domino Workflow (IBM/Lotus, [46]), Meteor (UGA/LSDIS, [56]), Mobile (UEN, [32]), MQ-Series/Workflow (IBM, [31]), Staffware (Staffware PLC, [59]), Verve Workflow (Versata, [61]), I-Flow (Fujitsu, [20]), InConcert (TIBCO, [60]), Changengine (HP, [29]), SAP R/3 Workflow (SAP, [54]), Eastman (Eastman, [57]), and FLOWer (Pallas Athena, [7]). In [6, 70] it is reported in what way each of these systems supports each of the patterns. For each product-pattern combination, we checked whether it is possible to realise the workflow pattern with the tool. Most of the products only offer direct support for less than half of the patterns. This does not imply that it is impossible to realise the patterns. In many cases, the designer can resort of coding or spaghetti-like diagrams (i.e., diagrams with many arcs to enumerate all possible combinations). Clearly, this makes such systems less suitable for workflow support. Some of the patterns can be realised by constructing them using other patterns. For example, Pattern 6 (Multi-choice) can be constructed from a combination of Pattern 4 (Exclusive choice) and Pattern 2 (Parallel split). The resulting diagram may be considerably larger, but it is fairly simple to realise. However, the more advanced synchronisation patterns, patterns involving a dynamic number of multiple instances, and the state-based patterns are more difficult, if not impossible, to realise if not present.

While the results reported in [6] mainly refer to suitability, the results in [36–38] consider formal expressiveness. In this work several “idealised” classes of workflow languages have been analyzed in terms of their expressive power. For example, MQ-Series/Workflow (IBM, [31]) is an example of a language beloning to the so-called class of *Synchronising Workflow Models*. Synchronising Workflow Models can be thought of as propagating true and false tokens. If an activity receives a true token, it will execute. If it receives a false token, it will simply pass it on. Branches of choices not chosen propagate false tokens, while branches that are chosen propagate true tokens. Synchronisation points await tokens from all incoming branches and, depending on their type, either 1) propagate a true token if it has received at least one true token, and a false token otherwise, or 2) propagate a true token if it has received only true tokens, and a false token otherwise. Synchronising Workflow Models offer direct support for Pattern

7 (Synchronising merge) but have problems dealing with loops. A detailed discussion of the formal expressiveness is beyond the scope of this paper. However, it is important to note that there are striking differences between languages and many of the more advanced constructs cannot be realised by the basic control-flow patterns.

The lessons learned by both evaluating contemporary systems using a set of workflow patterns and a detailed analysis of the fundamental control-flow mechanisms provide a solid basis for YAWL. The remainder of this paper is organised as follows. In Section 3 we analyse the suitability of Petri nets as a workflow language. Based on this and the lessons learned, we present the control-flow perspective of a new workflow language named YAWL in Section 4. We give a formal definition, provide formal semantics, and introduce a correctness notion. Using this correctness notion we will show that it can be verified in a compositional way. To conclude the paper, we relate YAWL to existing approaches, summarise the main results, and discuss future plans.

3 Limitations of Petri Nets as a control-flow language for workflows

Given the fact that workflow management systems have problems dealing with workflow patterns it is interesting to see whether established process modelling 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 colour (i.e., data), time, and hierarchy [5]. Examples of such languages are the coloured Petri nets as described in [34], the combination of Petri nets and Z specification described in [27], 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 [34] as much as possible.

Compared to existing languages high-level Petri nets are quite expressive when it comes to supporting the workflow patterns. 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 modelling effort is acceptable. 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 when it comes to modelling the control flow, 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 synchronisation 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 synchronised. 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 synchronised. However, at the end of the part which is instantiated multiple times all child cases having the same parent should be synchronised, i.e., case x can only continue if for each child case $x.1 \ldots x.n$ the part has been processed. In this synchronisation 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 synchronised in the right way. Clearly, a good workflow language does not put the burden of keeping track of these instances and synchronising them at the right level on the workflow designer.

Besides keeping track of identities and synchronising them at the right level, it is important to know how many child cases need to be synchronised. 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 synchronised. Clearly, it is also not acceptable to put the burden of modelling such a counter on the workflow designer.

3.2 Advanced synchronisation 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 synchronised, 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 synchronised. 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 synchronised. The situation where there is sometimes no synchronisation (XOR-join), sometimes full synchronisation (AND-join), and sometimes only partial synchronisation (OR-join) needed is referred to as Pattern 7 (Synchronising merge).

It is interesting to note that the Synchronising 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 Synchronising 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 synchronise the flows corresponding to these two elements. This can be done by putting a token in the input place of the synchronisation transition corresponding to the element car rental. Secondly, 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. Thirdly, 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 synchronisation 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 Synchronising merge has been handled in existing systems and literature. In the context of MQSeries Workflow the technique of “dead-path elimination” is used [41, 31]. 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 synchronisations was already raised in [23]. However, the bipolar synchronisation schemes presented in [23] are primarily aimed at avoiding constructs such as the Synchronising merge, i.e., the nodes are pure AND/XOR-splits/joins and partial synchronisation is not supported nor investigated. In the context of Event-driven Process Chains (EPC's, cf. [35]) the problem of dealing with the Synchronising merge also pops up. The EPC model allows for so-called $\vee$-connectors (i.e., OR-joins which only synchronise the flows that are active). The semantics of these $\vee$-connectors have been often debated [3, 11, 39, 51, 53]. In [3] the explicit modelling 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. [39] propose an approach based on Boolean tokens (case 2). Rump [53] proposes an intelligent scheduler to decide whether an $\vee$-connector should synchronise or not (case 3). In [51] three different join semantics are proposed for the $\vee$-connector: (1) *wait for all to come* (corresponds to the Synchronising 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 synchronisation problems in EPC's and workflow systems illustrates that patterns like the Synchronising merge are relevant and far from trivial.

3.3 Cancellation patterns

Most workflow modelling 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 than 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. Modelling 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 limitations of high-level Petri nets when it comes to

(1) patterns involving multiple instances, (2) advanced synchronisation 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 modelling 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 the preceding sections we have demonstrated that contemporary workflow management systems are less suitable and that high-level Petri nets, although providing a good starting point, do not solve all of these problems. This triggered the development of the language named *YAWL (Yet Another Workflow Language)*. The goal of this joint effort between Eindhoven University of Technology and Queensland University of Technology is to overcome the limitations mentioned in the previous section. The starting point will be Petri nets extended with constructs to address the multiple instances, advanced synchronisation, and cancellation patterns. In this section, we define the language and provide operational semantics.

4.1 Definition

Figure 2 shows the modelling elements of YAWL. YAWL extends the class of workflow nets described in [2, 5] with multiple instances, composite tasks, OR-joins, removal of tokens, and directly connected transitions. YAWL is inspired by Petri nets but is not just a macro package built on top of high-level Petri nets: It is a completely new language with independent semantics.[1]

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*[2] 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.

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

[1] Note that YAWL can be mapped onto high-level Petri nets. However, this mapping is far from trivial and YAWL can also be mapped to any other Turing complete language. Therefore, we would like to emphasise that the semantics of YAWL is independent from high-level Petri nets.

[2] Note that in YAWL we use the term *task* rather than *activity* to remain consistent with earlier work on workflow nets [2, 5].

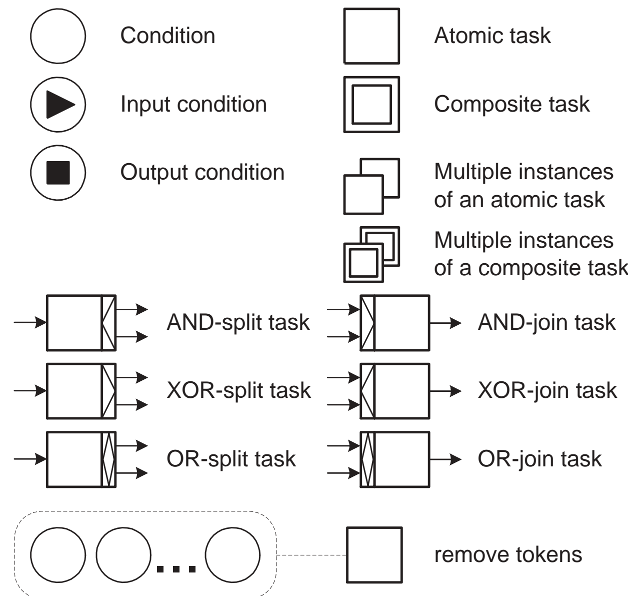


Fig. 2. Symbols used in YAWL.

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 initial instances. The value of the parameter is "static" if after creation no instances can be added and "dynamic" if it is possible to add additional instances while there are still instances being processed. Note that by extending Petri-nets with a construct having these four parameters (lower bound, upper bound, threshold, and static/dynamic), 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, 5] for AND/XOR-splits/joins as shown in Figure 2. Moreover, we introduce OR-splits and OR-joins corresponding respectively to Pattern 6 (Multi choice) and Pattern 7 (Synchronising merge), cf. Section 3.2.

Finally, we introduce a notation to remove tokens from places irrespective of how many tokens there are. As Figure 2 shows this is denoted by dashed rounded rectangles and 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 extension was also proposed in [9] for the purpose of modelling dynamic workflows.

A workflow specification is composed of one or more extended workflow nets (EWF-nets). Therefore, we first formalise the notion of an EWF-net.

Definition 1 (EWF-net). *An extended workflow net (EWF-net) N is a tuple $(C, \mathbf{i}, \mathbf{o}, T, F, \mathit{split}, \mathit{join}, \mathit{rem}, \mathit{nofi})$ such that:*

- *C is a set of conditions,*
- *$\mathbf{i} \in C$ is the input condition,*
- *$\mathbf{o} \in C$ is the output condition,*
- *T is a set of tasks,*
- *$F \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\}) \cup (T \times T)$ is the flow relation,*
- *every node in the graph $(C \cup T, F)$ is on a directed path from i to o,*
- *$\mathit{split} : T \to \{AND, XOR, OR\}$ specifies the split behaviour of each task,*
- *$\mathit{join} : T \to \{AND, XOR, OR\}$ specifies the join behaviour of each task,*
- *$\mathit{rem} : T \not\to \mathbb{P}(T \cup C \setminus \{\mathbf{i}, \mathbf{o}\})$ specifies the additional tokens to be removed by emptying a part of the workflow, and*
- *$\mathit{nofi} : T \not\to \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{\mathit{dynamic}, \mathit{static}\}$ specifies the multiplicity of each task (minimum, maximum, threshold for continuation, and dynamic/static creation of instances).*

The tuple (C, T, F) corresponds to a classical Petri net [49] where C (the set of conditions) corresponds to places, T (the set of tasks) corresponds to transitions, and F is the flow relation. There are however two differences. First of all, there are two special conditions/places: $\mathbf{i}$ and $\mathbf{o}$. Secondly, the flow relation also allows for direct connections between tasks/transitions. Note that the idea to have a special input condition/place $\mathbf{i}$ and a special output condition/place $\mathbf{o}$ has been adopted from the class of workflow nets [2, 5]. The four functions split, join, rem, and nofi specify the properties of each task. The first two functions (i.e., split and join) are used to specify whether a task is an AND/OR/XOR-split/join. The third function rem is a partial function specifying which parts of the net should be emptied. Emptying a part of an EWF-net is like removing tokens from a selected set of places.[3] Note that the range of rem includes tasks. $\mathbb{P}(T \cup C \setminus \{\mathbf{i}, \mathbf{o}\})$ is the set of all sets including conditions in $C \setminus \{\mathbf{i}, \mathbf{o}\}$ and tasks in T. Removing tokens from a task corresponds to aborting the execution of that task. However, if a task is a composite task, its removal implies the removal of all tokens it contains.[4] nofi is a partial function specifying the attributes related to multiple instances.

Whenever we introduce an EWF-net N we assume C, $\mathbf{i}$, $\mathbf{o}$, T, F, split, join, rem, and nofi defined as $N = (C, \mathbf{i}, \mathbf{o}, T, F, \mathit{split}, \mathit{join}, \mathit{rem}, \mathit{nofi})$. If ambiguity is possible, we use subscripts, i.e., C_N, $\mathbf{i_N}$, $\mathbf{o_N}$, T_N, F_N, split_N, join_N, rem_N, and nofi_N. We use $\pi_1(\mathit{nofi}(t))$ to refer to the minimal number of instances initiated, $\pi_2(\mathit{nofi}(t))$ to

[3] Note that we did not formalise the *token* concept in the context of YAWL. This is just a reference to tokens as they are used to represent states in a Petri net. This reference should support readers familiar with Petri nets. Other readers should just think of tokens as objects indicating the partial state of a process. We will formalise the concept of a state in Section 4.3.

[4] Note that in an EWF-net there are no composite tasks. Composite tasks are created by relating EWF-nets using the map function as shown in Definition 2.

refer to the maximal number of instances initiated, $\pi_3(\mathit{nofi}(t))$ is the threshold value (to terminate before all instances have completed), and $\pi_4(\mathit{nofi}(t))$ indicates whether it is possible to add instances while handling the other instances.

For convenience, we extend the functions rem and nofi in the following way. If $t \in T \setminus \mathit{dom}(\mathit{rem})$, then $\mathit{rem}(t) = \emptyset$. If $t \in T \setminus \mathit{dom}(\mathit{nofi})$, then $\pi_1(\mathit{nofi}(t)) = 1$, $\pi_2(\mathit{nofi}(t)) = 1$, $\pi_3(\mathit{nofi}(t)) = \infty$, $\pi_4(\mathit{nofi}(t)) = \mathit{static}$. This allows us to treat these partial functions as total functions in the remainder (unless we explicitly inspect their domains).

Now we define a workflow specification.[5] Recall that a workflow specification is composed of EWF-nets such that they form a tree-like hierarchy.

Definition 2. *A workflow specification S is a tuple $(Q, \mathit{top}, T^\diamond, \mathit{map})$ such that:*

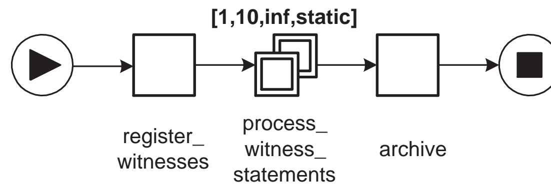
- *Q is a set of EWF-nets,*
- *$\mathit{top} \in Q$ is the top level workflow,*
- *$T^\diamond = \cup_{N \in Q} T_N$ is the set of all tasks,*
- *$\forall_{N_1, N_2 \in Q} N_1 \neq N_2 \Rightarrow (C_{N_1} \cup T_{N_1}) \cap (C_{N_2} \cup T_{N_2}) = \emptyset$, i.e., no name clashes,*
- *$\mathit{map} : T^\diamond \not\rightarrow Q \setminus \{\mathit{top}\}$ is a surjective injective function which maps each composite task onto an EWF net, and*
- *the relation $\{(N_1, N_2) \in Q \times Q \mid \exists_{t \in \mathit{dom}(\mathit{map}_{N_1})} \mathit{map}_{N_1}(t) = N_2\}$ is a tree.*

Q is a non-empty set of EWF-nets with a special EWF-net top. Composite tasks are mapped onto EWF-nets such that the set of EWF-nets forms a tree-like structure with top as root node. $T^\diamond$ is the set of all tasks. Tasks in the domain of map are composite tasks which are mapped onto EWF-nets. Throughout this paper we will assume that there are no name clashes, e.g., names of conditions differ from names of tasks and there is no overlap in names of conditions and tasks originating from different EWF-nets. If there are name clashes, tasks/conditions are simply renamed.

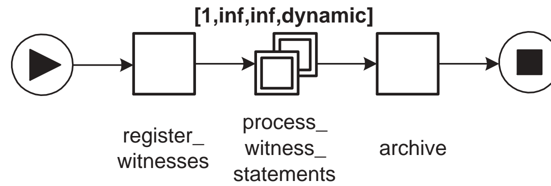
To illustrate the definitions in this section, we apply YAWL to some of the examples used in the previous section.

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_statements* after completing the initial task *register_witnesses*. 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 finishes if all have completed or at least three have completed. The three examples illustrate that YAWL allows for a direct specification of Multiple Instances With a Priori Runtime Knowledge (Pattern 14), Multiple Instances Without a Priori Runtime Knowledge (Pattern 15), and the Discriminator (Pattern 9).

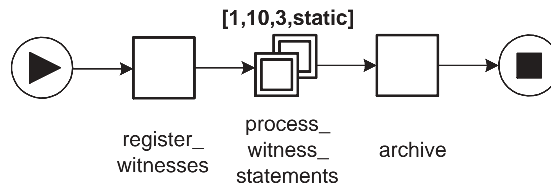
[5] Note that Definition 2 only refers to the control-flow perspective. Therefore, it should not be considered as a full specification of the workflow, e.g., data and organisational aspects are missing.



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



(b) A workflow processing an arbitrary number of witnesses with the possibility to add new witnesses "on the fly" (Pattern 15).



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

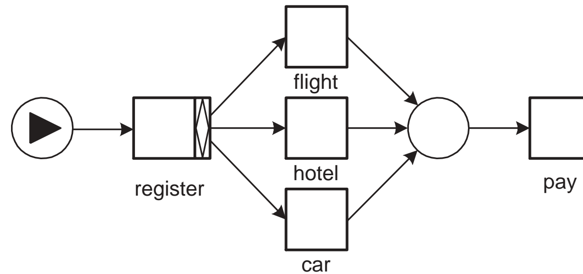
Fig. 3. Some examples illustrating the way YAWL deals with multiple instances.

Example: Advanced synchronisation 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., *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 synchronisation. However, if the trip contains two or even three elements, task *pay* is delayed until all have completed. This construct corresponds to the Synchronising 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 cancelled. 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 cancelled the moment the first one completes.

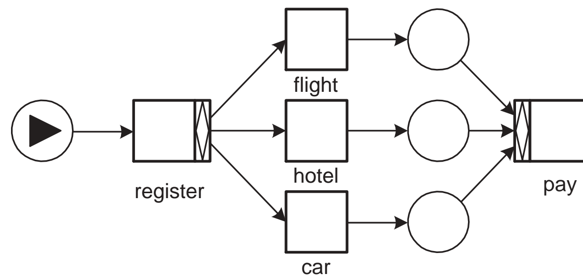
Example: Cancellation patterns Figure 5 illustrates the way YAWL supports the two cancellation patterns (patterns 19 and 20). The first workflow specification (a) shows the Cancel activity pattern which removes all tokens from the input conditions of task *activity*. In the second workflow specification (b) there is a task removing all tokens and putting a token in the output condition thus realising 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.

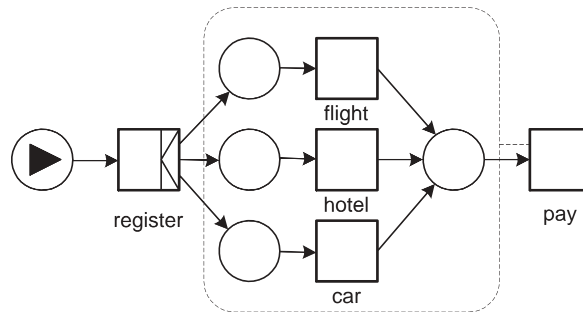
To conclude this section, we consider some more examples and discuss the role of constructs involving time. Figure 6 shows three fragments of a process where as a result of the booking of a flight, a hotel, and/or a car, task *pay* and/or task *cancel* are/is executed. The first workflow specification (a) starts with an AND-split *register* which enables tasks *flight*, *hotel* and/or *car*. The latter three tasks can have a positive or a negative outcome. Task *pay* can only be executed if all three booking tasks have a positive result. Task *cancel* can only occur if at least one of the booking tasks failed and the others have been completed. Note that in Figure 6(a) task *cancel* has to wait even if it is clear that there will never a payment. Moreover, tokens can get “stuck” in-between the three middle tasks (*flight*, *hotel* and *car*) and *pay*. The second workflow specification (b) improves this by making task *cancel* an XOR-join rather than an OR-join and removing all tokens from the relevant part of the process after cancellation. Figure 6(c) shows an even more sophisticated control flow. Unlike the first two specifications, the tasks *flight*, *hotel* and/or *car* are optional. Therefore, task *pay* can no longer be represented by an AND-join since not all three input tasks have to be executed. To solve this



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

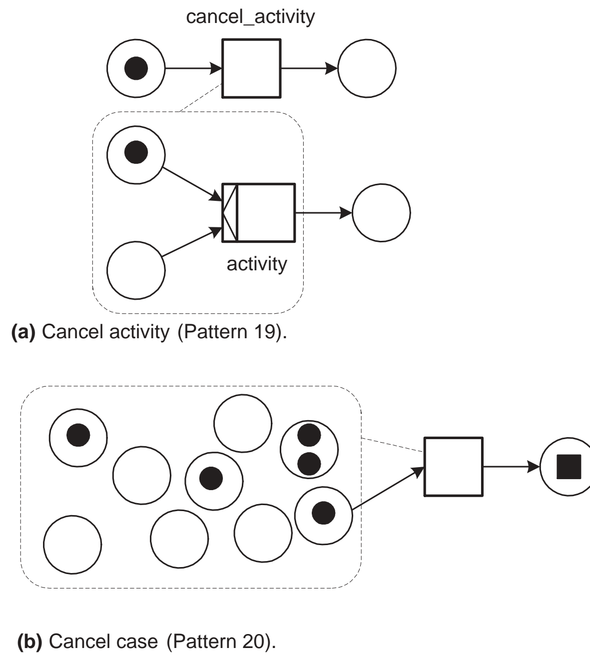


(b) Task pay is executed only once, i.e., after all started tasks have completed (Pattern 7).



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

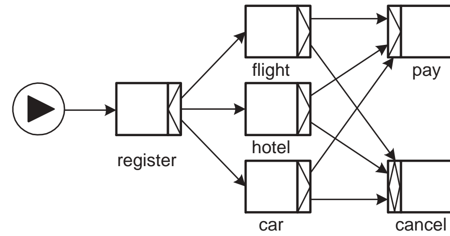
Fig. 4. Some examples illustrating the way YAWL deals with advanced synchronisation patterns.



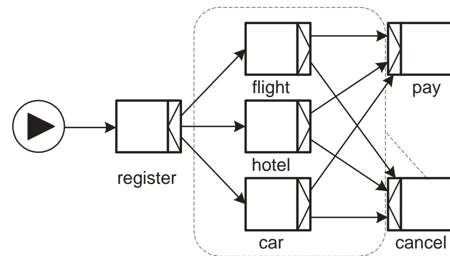
(a) Cancel activity (Pattern 19).

(b) Cancel case (Pattern 20).

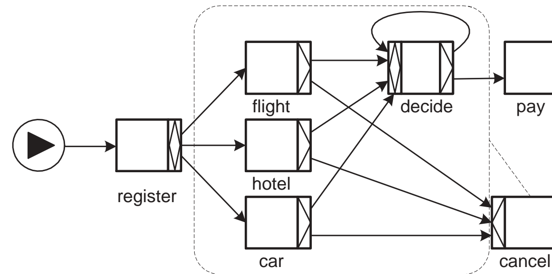
Fig. 5. Some examples illustrating the way YAWL deals with cancellation patterns.



(a) Task pay is executed if all three bookings succeed, otherwise cancel is executed after all booking attempts have completed.



(b) Task pay is executed if all three bookings succeed, if one of the bookings fails all other bookings are cancelled by task cancel.



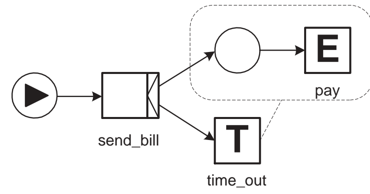
(c) Task cancel is executed if one of the bookings fails. Task decide will only be executed if no more bookings can succeed, and then decides whether to pay or wait for cancellation.

Fig. 6. Some more examples illustrating the more subtle behaviour of YAWL.

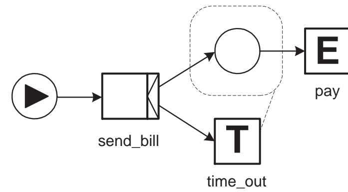
problem an additional task is introduced: *decide*. Task *decide* occurs only if all bookings that were enabled by *register* have been completed. Cancel may occur if one of the bookings failed. Note that both *decide* and *cancel* may be enabled at the same time. If *cancel* occurs before *decide*, there is no need to make a decision because one of the results was negative anyway. If *decide* occurs before *cancel*, it will decide to pay if all bookings were completed successfully. If one of the bookings failed, it should wait for the cancellation and therefore the one of the output arcs of *decide* also an input arc. This way the payment subprocess is fixed so that it can be cancelled. Figure 6 does not model any compensation, i.e., to cancel a booking it is often not sufficient to stop the control flow but one needs to initiate additional tasks to compensate previous steps in the process. It should be obvious that such aspects can be modelled by YAWL. [6]

Thus far, we do not consider the role of time and events. Clearly a workflow language should able to receive events and model timed constructs such as time-outs. Given the expressive power of YAWL we do not need to add explicit routing constructs for catching events and time-outs. We simply offer tasks which can handle these things. For example, we can offer various types of timed tasks, i.e., tasks that do nothing but wait for some specified time, and event tasks, i.e., tasks that do nothing other than waiting for an event to occur. For notational convenience we will label event tasks with the letter "E" and timed tasks with the letter "T". To clarify this consider Figure 7. In each of the four examples, task *pay* is an event task which indicates that it is an automatic task triggered by the occurrence of some event (e.g., the arrival of a message). Task *time_out* is a timed task, i.e., an automatic task that only waits for some time to complete. Since this is simply a task and not part of the YAWL language, there may be several predefined tasks offering such a service. The delay (i.e., the time the task takes to complete) may be absolute ("Wait until the due date of the order.") or relative ("Wait for 1 week from now.") and may depend on data or external conditions. In all four examples, there is a "race" between the payment and the time-out which starts after sending the bill. Figure 7(a) shows the situation where task *time_out* cancels the payment process when it completes. Note that even if the payment is being processed, the top branch may be cancelled. Also note that the condition between *send_bill* and *pay* is shown explicitly. This way it is possible to clearly indicate that also waiting payments are withdrawn when *time_out* completes. Figure 7(b) shows the situation where task *time_out* only cancels the payment process when it did not start yet. In Figure 7(a) and (b) the time-out process does not need to be cancelled after processing the payment because if the time-out occurs after payment there is nothing to cancel. However, if the task *time_out* does not just withdraw tokens but also represents some real work or triggers other tasks to further handle the cancellation process, then task *pay* needs to cancel the cancellation process as is shown in Figure 7(c). Task *cancel_order* is a task initiated after the time-out occurred to compensate the effect of not receiving the payment within the prespecified time. When task *pay* completes it cancels the whole lower branch. Note that *time_out* has an explicit input condition to indicate that if, for some reason, the task was not started when the payment is completed, the cancellation process is still cancelled properly. Figure 7(d) shows an extension of (c) where before

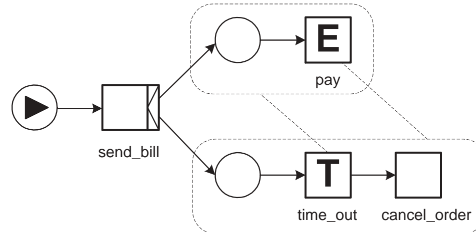
[6] Another approach would be to extend YAWL with transactional features (cf. Section 6). For the moment, we choose not to do so.



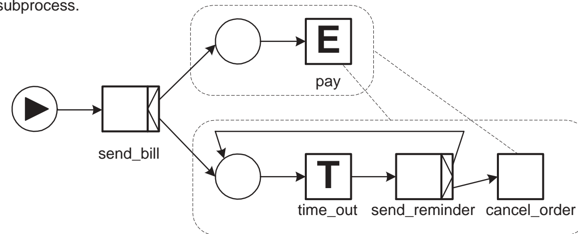
(a) Task time_out is a timed activity, if it finishes before task pay is completed it cancels the payment subprocess .



(b) The payment process is only cancelled if the time-out occurs before task pay is started.



(c) As in (a) but now the payment cancels the cancellation subprocess.



(d) Instead of immediately cancelling the payment process when the time-out completes, a prespecified number of reminders is sent before cancelling the order.

Fig. 7. Some more examples illustrating that YAWL can deal with constructs involving time and external events without offering dedicated constructs.

cancellation a couple of reminders are sent to the customer. Task *send_reminder* is an XOR-split which, based on the number of reminders already sent, decides to go for another reminder or cancel the payment process. Note that this way it is possible to model things like: "If the customer does not pay after one week, a reminder is sent. This is repeated four times or until the customer pays. If the customer did not pay after four reminders, the order is cancelled.".

The four examples shown in Figure 7 demonstrate that YAWL does not need specific constructs for patterns involving time and/or events. They also illustrate the ability of YAWL to specify complex workflow processes with challenging control-flow requirements.

4.2 Notation

Before describing the formal semantics of YAWL, we introduce some useful notations.

To navigate through an EWF-net it is useful to define the preset and postset of a node (i.e., either a condition or task). To simplify things we add an implicit condition $c_{(t_1,t_2)}$ between two tasks t_1, t_2 if there is a direct connection from t_1 to t_2. For this purpose we define the extended set of conditions C^{ext} and the extended flow relation F^{ext}.

Definition 3. *Let* $N = (C, \mathbf{i}, \mathbf{o}, T, F, \mathit{split}, \mathit{join}, \mathit{rem}, \mathit{nofi})$ *be an EWF-net.* $C^{ext} = C \cup \{c_{(t_1,t_2)} \mid (t_1,t_2) \in F \cap (T \times T)\}$ *and* $F^{ext} = (F \backslash (T \times T)) \cup \{(t_1, c_{(t_1,t_2)}) \mid (t_1,t_2) \in F \cap (T \times T)\} \cup \{(c_{(t_1,t_2)}, t_2) \mid (t_1,t_2) \in F \cap (T \times T)\}$. *Moreover, auxiliary functions* $\bullet_, _\bullet : (C^{ext} \cup T) \rightarrow \mathbb{P}(C^{ext} \cup T)$ *are defined that assign to each node its preset and postset, respectively. For any node* $x \in C^{ext} \cup T$, $\bullet x = \{y \mid (y,x) \in F^{ext}\}$ *and* $x \bullet = \{y \mid (x,y) \in F^{ext}\}$.

Note that the preset and postset functions depend on the context, i.e., the EWF-net the function applies to.

Definition 4. *Whenever we introduce a workflow specification* $S = (Q, \mathit{top}, T^\diamond, \mathit{map})$, *we assume* T^A, T^C, T^{SI}, T^{MI}, $C^\diamond$ *to be defined as follows:*

- $T^A = \{t \in T^\diamond \mid t \notin \mathit{dom}(\mathit{map})\}$ *is the set of atomic tasks,*
- $T^C = \{t \in T^\diamond \mid t \in \mathit{dom}(\mathit{map})\}$ *is the set of composite tasks,*
- $T^{SI} = \{t \in T^\diamond \mid \forall_{N \in Q}\, t \notin \mathit{dom}(\mathit{nofi}_N)\}$ *is the set of single instance tasks,*
- $T^{MI} = \{t \in T^\diamond \mid \exists_{N \in Q}\, t \in \mathit{dom}(\mathit{nofi}_N)\}$ *is the set of (potentially) multiple instance tasks, and*
- $C^\diamond = \cup_{N \in Q}\, C_N^{ext}$ *is the extended set of all conditions.*

If ambiguity is possible, we use subscripts, i.e., T_S^A, T_S^C, T_S^{SI}, T_S^{MI}, and $C_S^\diamond$. Within the context of a single workflow specification we will omit these subscripts. Moreover, since the domains of the functions split_N, join_N, rem_N, and nofi_N do not overlap for different $N \in Q$ we can omit the subscripts.

A workflow specification defines a tree-like structure. To navigate through this structure we define the function unfold. Given a set of nodes (i.e., tasks and conditions), unfold returns these nodes and all child nodes.

Definition 5. *Let $S = (Q, top, T^\diamond, map)$ be a workflow specification. We define the function unfold : $\mathbb{P}(T^\diamond \cup C^\diamond) \rightarrow \mathbb{P}(T^\diamond \cup C^\diamond)$ as follows. For $X \subseteq T^\diamond \cup C^\diamond$:*

$$unfold(X) = \begin{cases} \emptyset & \text{if } X = \emptyset \\ \{x\} \cup unfold(X \setminus \{x\}) & \text{if } x \in X \cap (C^\diamond \cup T^A) \\ \{x\} \cup unfold((X \setminus \{x\}) \cup T_{map(x)} \cup C^{ext}_{map(x)}) & \text{if } x \in X \cap T^C \end{cases}$$

Note that the $unfold(X)$ returns each node in X and all nodes contained by the nodes in X. For atomic tasks and conditions, no unfolding is needed. For composite tasks, all nodes contained by these tasks are included in the result, i.e., $unfold(X)$ recursively traverses all composite tasks in X.

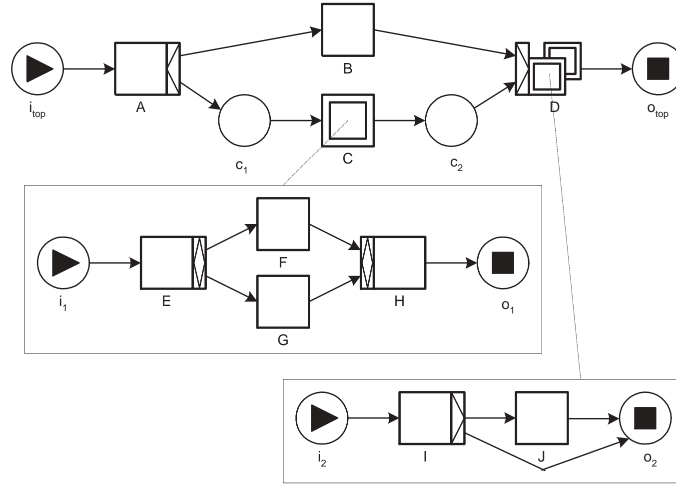


Fig. 8. An example.

Figure 8 shows an example of a workflow specification to illustrate the notations. $T^A = \{A, B, E, F, G, H, I, J\}$ is the set of atomic tasks and $T^C = \{C, D\}$ is the set of composite tasks. $T^{SI} = \{A, B, C, E, F, G, H, I, J\}$ and $T^{MI} = \{D\}$, i.e., D is the only task (potentially) having multiple instances. Apart from input and output conditions, there are only two explicit conditions (c_1 and c_2). All other conditions in the set $C^\diamond$ are implicit, e.g., $c_{(A,B)}$ is an implicit condition corresponding to the arc connecting A and B. $unfold(\{B, c_2, D\}) = \{B, c_2, D, i_2, I, J, o_2, c_{(I,J)}\}$, i.e., if D is unfolded and all tasks and conditions (including the implicit ones) contained by D are added.

4.3 Semantics

Definition 2 defines, in mathematical terms, the syntax of a workflow specification. Based on this definition it is straightforward to give a concrete workflow language, e.g.

in terms of XML. However, Definition 2 does not give any semantics. Thus far we have only given intuitive descriptions of the dynamic behaviour of a workflow specification S. In the remainder of this section we will provide a formal semantics. We will do this in three steps. First, we introduce some preliminaries (bags and identifiers). Then we define the state space corresponding to a workflow specification S. Finally, we specify the state transitions possible.

Preliminaries The definition of the state space corresponding to a workflow specification is inspired by the concept of tokens in coloured Petri nets. The state space consists of a collection of tokens having a value. Since we abstract from data in this paper, it suffices that each token has an *identity*. To deal with multiple tokens in the same place having the same identity, we need to resort to *bags*. Moreover, we need to structure the set of case identifiers to relate child instances to parent instances. The latter is needed to deal with multiple instances.

In this paper, bags are defined as finite multi-sets of elements from some alphabet A. A bag over alphabet A can be considered as a function from A to the natural numbers $\mathbb{N}$ such that only a finite number of elements from A are assigned a non-zero function value. For some bag X over alphabet A and $a \in A$, $X(a)$ denotes the number of occurrences of a in X, often called the cardinality of a in X. The set of all bags over A is denoted $\mathcal{B}(A)$. For the explicit enumeration of a bag, a notation similar to the notation for sets is used, but using square brackets instead of curly brackets and using superscripts to denote the cardinality of the elements. For example, $[a^2, b, c^3]$ denotes the bag with two elements a, one b, and three elements c; the bag $[a^2 \mid P(a)]$ contains two elements a for every a such that $P(a)$ holds, where P is some predicate on symbols of the alphabet under consideration. To denote individual elements of a bag, the same symbol "$\in$" is used as for sets: For any bag X over alphabet A and element $a \in A$, $a \in X$ if and only if $X(a) > 0$. The sum of two bags X and Y, denoted $X \uplus Y$, is defined as $[a^n \mid a \in A \wedge n = X(a) + Y(a)]$. The difference of X and Y, denoted $X - Y$, is defined as $[a^n \mid a \in A \wedge n = \max((X(a) - Y(a)), 0)]$. $\mathit{size}(X) = \sum_{a \in A} X(a)$ is the size of the bag. The binding of sum and difference is left-associative. The restriction of X to some domain $D \subseteq A$, denoted $X \upharpoonright D$, is defined as $[a^{X(a)} \mid a \in D]$. Restriction binds stronger than sum and difference. The notion of subbags is defined as expected: Bag X is a subbag of Y, denoted $X \subseteq Y$, if and only if, for all $a \in A$, $X(a) \leq Y(a)$. $X \subset Y$, if and only if, $X \subseteq Y$ and for some $a \in A$, $X(a) < Y(a)$. Note that any finite set of elements from A also denotes a unique bag over A, namely the function yielding 1 for every element in the set and 0 otherwise. Therefore, finite sets can also be used as bags. If X is a bag over A and Y is a finite subset of A, then $X - Y$, $X \uplus Y$, $Y - X$, and $Y \uplus X$ yield bags over A. Moreover, $X \subseteq Y$ and $Y \subseteq X$ are defined in a straightforward manner.

Less straightforward is the way we deal with *case identifiers*. Each case (i.e., workflow instance) needs to have a unique identifier. Moreover, parts of the process may be instantiated multiple times. These sub-instances should again have unique identifiers, etc. To handle these issues, we assume an infinite set I extended with parent-child relationships.

Definition 6. *I is an infinite set of case identifiers. We define the following functions on I:*

- *$child : I \times \mathbb{N} \setminus \{0\} \rightarrow I$. $child(i, n)$ is the n-th child of i.*
- *$children : I \rightarrow \mathbb{P}(I)$. $j \in children(i)$ if and only if j is a child of i, i.e., $children(i) = \{child(i, n) \mid n \in \mathbb{N} \setminus \{0\}\}$.*
- *$children^* : I \rightarrow \mathbb{P}(I)$ is the reflexive transitive closure of children, i.e., $children^*(i) = \cup_{n \in \mathbb{N}}\ children^n(i)$, where $children^0(i) = \{i\}$ and $children^{n+1}(i) = children(children^n(i))$.*[7]

Function child is defined such that for any $i, j \in I$ and $n, m \in \mathbb{N}$: $child(i, n) = child(j, m)$ implies $i = j$ and $n = m$.

Each identifier has an infinite number of child instances. These child instances are ordered and non-overlapping, i.e., $child(i, n)$ is the n-th child of i and for any $i, j \in I$ and $n, m \in \mathbb{N}$: $child(i, n) = child(j, m)$ implies $i = j$ and $n = m$. Based on $child$ the functions $children$ and $children^*$ provide the set of direct children and the set of all descendants respectively. Note that $i \in children^*(i)$ and $i \notin children(i)$ for any $i \in I$. It is possible to construct I and the corresponding functions as is shown in Chapter 11 of [27]. An example of an encoding of I is $I = \mathbb{N}^*$ (sequences of natural numbers) and $child(i, n) = i.n$ where $i \in \mathbb{N}^*$ and $n \in \mathbb{N} \setminus \{0\}$. $62.231.77$ is an element of such I, $child(62.231.77, 9) = 62.231.77.9$, $children(62.231.77) = \{62.231.77.1, 62.231.77.2, \ldots\}$, and $children^*(62.231.77) = \{62.231.77, 62.231.77.1, 62.231.77.2, \ldots, 62.231.77.1.1, 62.231.77.1.2, \ldots\}$. Note that such encoding is similar to IP addresses on the Internet.

Using the notation for bags and the definition of I we can define the state space of a workflow specification.

State space We will use the set of case identifiers I to distinguish cases. Moreover, because of parallelism, a single case can be in multiple conditions and/or tasks. Therefore, we represent a state as a bag of objects where each object is represented by a location and an identity. Readers familiar with (coloured) Petri nets can think of these objects as (coloured) tokens. Readers not familiar with Petri nets can think of these objects as "substates" or "threads of control". In the remainder we will use the term token.[8] For reasons of simplicity we will assume that each task has four states: $exec_t$ (for a task (instance) being executed), mi_a_t (for a task (instance) being active), mi_e_t (for a task being entered, i.e., task instances which have been created but not yet being executed), and mi_c_t (for a task (instance) that has completed). The states mi_a_t, mi_e_t, and mi_c_t have been added to deal with multiple instances. If there would not be multiple instances, the state $exec_t$ would have been sufficient. State mi_e_t keeps track of task instances that have been created but still are waiting to be executed. State mi_c_t keeps track of task instances that have been completed. Task instances move from mi_e_t, to

[7] Note that we have applied the function $children$ to a set of identifiers rather than a single element. This extended use, however, is straightforward.

[8] In spite of the use of Petri-net terminology, it is important to note that we do not use standard Petri-net semantics, e.g., the firing rule is modified considerably.

$exec_t$, to mi_c_t. For any token residing in any of these three states there is a corresponding token in state mi_a_t to keep track of all instances.

Definition 7. *A workflow state s of a specification $S = (Q, top, T^\diamond, map)$ is a multiset over $Q^\diamond \times I$ where $Q^\diamond = C^\diamond \cup (\cup_{t \in T^\diamond} \{exec_t, mi_a_t, mi_e_t, mi_c_t\})$, i.e., $s \in \mathcal{B}(Q^\diamond \times I)$.*

A workflow state s is a bag of tokens where each token is represented by a pair consisting of a condition from $Q^\diamond$ and an identifier from I, i.e., $s \in \mathcal{B}(Q^\diamond \times I)$. For a token $(x, i) \in s$, x denotes the *location* of the token and i denotes the *identity* of the token. Location x is either (1) an implicit or explicit condition ($x \in Q^\diamond$) or (2) a task state of some task $t \in T^\diamond$ ($x \in \{exec_t, mi_a_t, mi_e_t, mi_c_t\}$). When defining the state transitions it will become clear that reachable workflow states will satisfy the invariant that the number of tokens in mi_a_t equals the sum of tokens in $exec_t$, mi_e_t, and mi_c_t, i.e., the number of active instances equals the number of executing, entered, and completed instances.

Note that in the workflow state we do not distinguish between atomic and composite tasks. We could have omitted task states for composite tasks. For reasons of simplicity, we did not do so. For similar reasons we do not distinguish between tasks having a single instance and tasks (potentially) having multiple instances. We could have omitted mi_a_t, mi_e_t, and mi_c_t for each $t \in T^{SI}$. However, this would have complicated things without changing the semantics. Instead we assume that for $t \in T^{SI}$: $\pi_1(nofi(t)) = 1$, $\pi_2(nofi(t)) = 1$, $\pi_3(nofi(t)) = \infty$, $\pi_4(nofi(t)) = static$.

To query workflow states (or parts of workflow states), we define three projection functions.

Definition 8. *Let $s \in \mathcal{B}(Q^\diamond \times I)$ be a workflow state and $x \in Q^\diamond$. $id(s) = \{i \mid (y, i) \in s\}$, $id(s, x) = [i^n \mid i \in I \wedge n \in \mathbb{N} \wedge n = \sum_{(x,i) \in s} s(x, i)]$, and $q(s) = [y^n \mid y \in Q^\diamond \wedge n \in \mathbb{N} \wedge n = \sum_{i \in I \mid (y,i) \in s} s(y, i)]$.*

$id(s)$ returns the set of all identities appearing in state s. $id(s, x)$ returns a bag of identities appearing in location x in state s. $q(s)$ returns a bag of locations marked in state s by some token.

State transitions To complete the semantics of a workflow specification, we need to specify all possible state transitions. The state space combined with the transition relation defines a transition system.

Figure 9 shows all possible transitions for an atomic task t using a Petri-net-like notation.[9] There are five types of transitions: $enter$, $start$, $complete$, $exit$, and add. Each of these transitions is relative to some task t. Figure 9 also shows the four task states mi_a_t, $exec_t$, mi_e_t, and mi_c_t. The decomposition of an atomic task into five transitions is independent of the hierarchical structure of the model. Therefore, Figure 9 should not be confused with the YAWL diagrams shown before. It should only be considered as a “roadmap” for reading the subsequent definitions.

[9] Although some of the terminology (e.g. token and transition) is borrowed from Petri nets, the diagram should not be read as a normal Petri net, e.g., $enter$ produces a variable number of tokens and $exit$ consumes a variable number of tokens as indicated by the “thick” arcs.

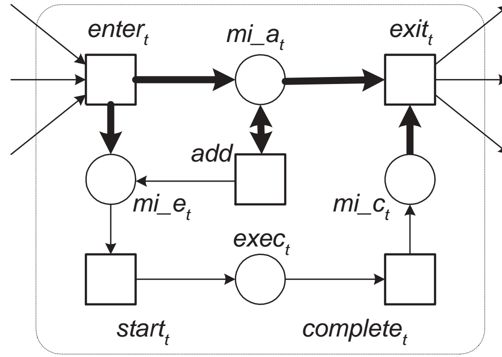


Fig. 9. Illustration of the semantics of an atomic task t.

Transition $enter$ occurs when it is time to execute task t. If $join(t) = XOR$, it suffices if one of the conditions in the preset of t holds a token. If $join(t) = AND$, each of the input conditions needs to hold a token corresponding to the same case/instance (i.e., all input tokens need to have the same identity). If $join(t) = OR$, the number of input tokens is in-between the number of tokens needed for the XOR-join and AND-join. (We will come back to this later.) As in an ordinary Petri net, transition $enter$ removes the tokens that enabled the task. The number of tokens produced depends on the number of instances that needs to be created. If $t \in T^{SI}$, two tokens are produced: one for mi_a_t and one for mi_e_t. If more instances need to be created, more tokens are produced. However, the number of tokens produced for mi_a_t always matches the number of tokens produced for mi_e_t. Similarly, the identities match.

Transition $start$ occurs when the execution of the task starts. This transition is enabled for each token in mi_e_t. When this transition fires it consumes a token from mi_e_t and produces a token for $exec_t$.

Transition $complete$ occurs when the execution is completed. When this transition fires it consumes a token from $exec_t$ and produces a token for mi_c_t.

Transition $exit$ occurs when all instances corresponding to the same parent have completed. This transition is not like an ordinary Petri-net transition in the sense that the number of tokens consumed depends on the number of instances created. In addition, this transition removes tokens from selected parts of the specification as indicated by $rem(t)$. The number of tokens produced depends on $split(t)$.

Transition add is only relevant for tasks t with $t \in T^{MI}$ and $\pi_4(nofi(t)) = dynamic$. As long as the maximum number is not reached, the transition can create new instances by adding a token to mi_e_t. The transition removes all tokens having the same parent from mi_a_t and returns these tokens to mi_a_t including an additional token corresponding to the new child instance. This may seem to be a complicated way of adding a single token. However, it is done this way to make sure that the token gets the right identifier.

Figure 9 only considers atomic tasks. If t is a composite task, there is a corresponding subnet $map(t)$. Figure 10 illustrates how the composite task t is connected

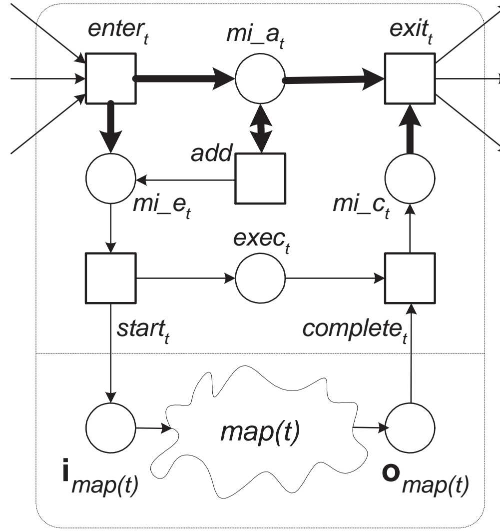


Fig. 10. Illustration of the semantics of a composite task t.

through the input and output conditions ($\mathbf{i}_{map(t)}$ and $\mathbf{o}_{map(t)}$) to the subnet $map(t)$. At the semantical level each composite task is decomposed into the structure shown in Figure 10. If $t \in T^C$, then $start$ produces an additional token for the input condition of the corresponding subnet. Moreover, if $t \in T^C$, then $complete$ can only fire if the output condition of the corresponding subnet holds a token with the right identity (i.e., the identities of the tokens in $exec_t$ and $\mathbf{o}_{map(t)}$ match). Again we would like to stress that Figure 10 is not a YAWL model, it is only used to illustrate the semantics which will be defined in this section. Workflow designers using YAWL do not need to see this internal behaviour. Moreover, it is important to note that although YAWL borrows some of the concepts of Petri nets it is not some extension built on top of high-level Petri nets. It is a completely new language having independent semantics as will be shown in the remainder.

The following definitions formalise the set of transitions possible in a given state by specifying so-called *binding relations*. The first binding relation is $binding_{enter}$. $binding_{enter}(t, i, c, p, s)$ is a Boolean expression which evaluates to true if $enter$ can occur for task t, case/instance i, in state s, while consuming the bag of tokens c and producing the bag of tokens p.

Definition 9. *Let $S = (Q, top, T^\diamond, map)$ be a specification and $t \in T^\diamond$, $i \in I$, $c, p, s \in \mathcal{B}(Q^\diamond \times I)$. The Boolean function $binding_{enter}(t, i, c, p, s)$ yields true if and only if an $n \in \mathbb{N}$ exists such that* all *of the following conditions hold:*

– *Tokens to be consumed are present in the state:*

$$c \subseteq s$$

– *Tokens are consumed from the input conditions of the task involved and at most one token can be consumed from each condition of the preset:*

$$q(c) \subseteq \bullet t$$

– n *tokens are created both for the* active task state *and the* entered task state *of the task involved:*

$$p = [(mi_a_t, j) \mid j = child(i,k) \wedge 1 \leq k \leq n] \uplus$$
$$[(mi_e_t, j) \mid j = child(i,k) \wedge 1 \leq k \leq n]$$

– *The number* n *of tokens created is not less than the lower bound and not more than the upper bound specified for the task involved:*

$$\pi_1(nofi(t))) \leq n \leq \pi_2(nofi(t))$$

– *Tokens to be produced were not already in the state, i.e., a task is instantiated only once for a given identifier:*

$$s \cap p = \emptyset$$

– *Tokens to be consumed all refer to the same identity:*

$$\{i\} = id(c)$$

– *For AND-join behaviour, all input conditions need to have tokens:*

$$join(t) = AND \Rightarrow q(c) = \bullet t$$

– *For OR-join behaviour, at least one input condition needs to have tokens:*

$$join(t) = OR \Rightarrow q(c) \neq \emptyset$$

– *For XOR-join behaviour, only one input condition should have a token, and, in addition, this input condition should not have more than one token:*

$$join(t) = XOR \Rightarrow q(c) \textit{ is a singleton}$$

Similarly, the other binding relations are given and $binding(t, i, c, p, s)$ is the overall binding relation. The next binding relation is $binding_{start}$.

Definition 10. *Let* $S = (Q, top, T^\diamond, map)$ *be a specification and* $t \in T^\diamond$, $i \in I$, $c, p, s \in \mathcal{B}(Q^\diamond \times I)$. *The Boolean function* $binding_{start}(t, i, c, p, s)$ *yields true if and only if* all *of the following conditions hold:*

– *Tokens to be consumed are present in the state:*

$$c \subseteq s$$

– *Just one token is consumed which was in the* entered task state *of the task involved:*

$$c = [(mi_e_t, i)]$$

– *A token is produced for the input condition of the corresponding decomposition (if existing) and a token is produced for the* executing task state *of the task involved:*

$$p = [(exec_t, i)] \uplus [(\mathbf{i}_{map(t)}, i) \mid t \in T^C]$$

The counterpart of $binding_{start}$ is $binding_{complete}$.

Definition 11. *Let* $S = (Q, top, T^\diamond, map)$ *be a specification and* $t \in T^\diamond$*,* $i \in I$*,* $c, p, s \in \mathcal{B}(Q^\diamond \times I)$*. The Boolean function* $binding_{complete}(t, i, c, p, s)$ *yields true if and only if* all *of the following conditions hold:*

– *Tokens to be consumed are present in the state:*

$$c \subseteq s$$

– *For a token to be able to complete, it has to be in the* executing task state *of the task involved and its decomposition (if present) needs to be finished:*

$$c = [(exec_t, i)] \uplus [(\mathbf{o}_{map(t)}, i) \mid t \in T^C]$$

– *Completing implies producing a token for the* completed task state *of the task involved:*

$$p = [(mi_c_t, i)]$$

Next we define the binding relation for the $exit$ transition. This is done in two steps. $binding_{exit}(t, i, c, p, s)$ (Definition 12) does not take the removal of additional tokens into account, i.e., $rem(t)$ is ignored. However, $binding^{rem}_{exit}(t, i, c, p, s)$ (Definition 13) extends $binding_{exit}(t, i, c, p, s)$ to remove these additional tokens.

Definition 12. *Let* $S = (Q, top, T^\diamond, map)$ *be a specification and* $t \in T^\diamond$*,* $i \in I$*,* $c, p, s \in \mathcal{B}(Q^\diamond \times I)$*. The Boolean function* $binding_{exit}(t, i, c, p, s)$ *yields true if and only if* all *of the following conditions hold:*

– *Tokens to be consumed are present in the state:*

$$c \subseteq s$$

– *Tokens are produced only for output conditions of the task involved and at most one token can be produced for each condition in the postset:*

$$q(p) \subseteq t\bullet$$

– *Tokens to be consumed are children of the instance considered, and they occur in both the* active task state *and the* completed task state *of the task involved (and tokens in other task states are not affected):*

$$c \subseteq \uplus\{[(mi_a_t, j), (mi_c_t, j)] \mid j \in children(i)\}$$

– *If a token is consumed from the* active task state *then its corresponding token is removed from the* completed task state *and vice versa:*

$$id(c, mi_a_t) = id(c, mi_c_t)$$

- *All children of the identifier involved have completed or at least as many as required by the threshold of the task involved have completed:*

$$id(s - c, mi_a_t) \cap children(i) = \emptyset \vee size(id(c, mi_c_t)) \geq \pi_3(nofi(t))$$

- *The only tokens produced are those with the identifier involved:*

$$\{i\} = id(p)$$

- *For AND-split behaviour, tokens are produced for all output conditions of the task involved:*

$$split(t) = AND \Rightarrow q(p) = t\bullet$$

- *For OR-split behaviour, tokens are produced for some of the output conditions of the task involved:*

$$split(t) = OR \Rightarrow q(p) \neq \emptyset$$

- *For XOR-split behaviour, a token is produced for exactly one of the output conditions of the task involved:*

$$split(t) = XOR \Rightarrow q(p) \textit{ is a singleton}$$

Definition 13. *Let* $S = (Q, top, T^\diamond, map)$ *be a specification and* $t \in T^\diamond$*,* $i \in I$*,* $c, p, s \in \mathcal{B}(Q^\diamond \times I)$*. The Boolean function* $binding^{rem}_{exit}(t, i, c, p, s)$ *yields true if and only if a* $c' \in \mathcal{B}(Q^\diamond \times I)$ *exists with* $binding_{exit}(t, i, c', p, s)$ *such that:*

$$\begin{aligned} c = c' \uplus [(x, j) \in s - c' \mid \ & j \in children^*(i) \wedge \\ & ((\exists_{t' \in unfold(rem(t) \cup \{t\}) \cap T^\diamond} \\ & x \in \{exec_{t'}, mi_a_{t'}, mi_e_{t'}, mi_c_{t'}\}) \vee \\ & \ x \in unfold(rem(t) \cup \{t\}) \cap C^\diamond)] \end{aligned}$$

I.e., from all conditions part of task t (and its descendants, if it has a decomposition) and part of its removal set (and descendants of elements thereof) tokens are removed that correspond to children of identifier i*. Note that in case the threshold for continuation was reached, executing the* exit *part of the task involved implies cancellation of all child instances (if existing) which have not yet completed.*

Definition 14. *Let* $S = (Q, top, T^\diamond, map)$ *be a specification and* $t \in T^\diamond$*,* $i \in I$*,* $c, p, s \in \mathcal{B}(Q^\diamond \times I)$*. The Boolean function* $binding_{add}(t, i, c, p, s)$ *yields true if and only if an* $n \in \mathbb{N}$ *exists such that* all *of the following conditions hold:*

- *Tokens to be consumed are present in the state:*

$$c \subseteq s$$

- *The tokens to be consumed are children of the identifier involved present in the* active task state *of the task involved:*

$$c = [(mi_a_t, j) \mid j = child(i, k) \wedge 1 \leq k \leq n]$$

– *All children of the identifier involved present in the* active task state *of the task involved are consumed:*

$$id(s - c, mi_a_t) \cap children(i) = \emptyset$$

– *It is still possible to create an extra token without violating the upper bound specified for the task involved:*

$$n < \pi_2(nofi(t))$$

– *The task involved should allow for the creation of extra tokens while handling the other tokens:*

$$\pi_4(nofi(t)) = dynamic$$

– *An extra token is added to the* active task state *and to the* entered task state *of the task involved:*

$$p = c \uplus [(mi_a_t, child(i, n + 1)), (mi_e_t, child(i, n + 1))]$$

Note that n tokens are removed from mi_a_t and $n + 1$ are returned to this task state. This way the result is one additional token in the active task state. The same token is added to the entered task state mi_e_t.

Now we can define the binding relation $binding(t, i, c, p, s)$. Note that this relation excludes OR-joins. OR-joins will be incorporated later.

Definition 15. *Let $S = (Q, top, T^\diamond, map)$ be a specification and $t \in T^\diamond$, $i \in I$, $c, p, s \in \mathcal{B}(Q^\diamond \times I)$. The Boolean function $binding(t, i, c, p, s)$ yields true if and only if* any *of the following conditions holds:*

– *The* enter *part of a task is enabled, but the task does not have OR-join behaviour:*

$$binding_{enter}(t, i, c, p, s) \wedge join(t) \neq OR$$

– *The* start *part of a task is enabled:*

$$binding_{start}(t, i, c, p, s)$$

– *The* complete *part of a task is enabled:*

$$binding_{complete}(t, i, c, p, s)$$

– *The* exit *part of a task is enabled:*

$$binding_{exit}^{rem}(t, i, c, p, s)$$

– *The* add *part of a task is enabled:*

$$binding_{add}(t, i, c, p, s)$$

Based on the explanations given before, the definitions of most bindings are straightforward. As indicated before, $binding^{rem}_{exit}(t, i, c, p, s)$ extends $binding_{exit}(t, i, c, p, s)$ and the overall binding relation $binding(t, i, c, p, s)$ excludes OR-joins. It is important to note that Figure 10 is just added for illustration purposes: The transition rules are quite different from the standard firing rule in a (coloured) Petri net.

Using $binding(t, i, c, p, s)$ we can define a partial transition relation excluding OR-joins.

Definition 16. *Let $S = (Q, top, T^{\diamond}, map)$ be a specification and s_1 and s_2 two workflow states of S. $s_1 \rightarrowtail s_2$ if and only if there are $t \in T^{\diamond}$, $i \in I$, $c, p \in \mathcal{B}(Q^{\diamond} \times I)$ such that $binding(t, i, c, p, s_1)$ and $s_2 = (s_1 - c) \uplus p$.*

$\rightarrowtail$ defines a partial transition relation on the states of workflow specification. The reflexive transitive closure of $\rightarrowtail$ is denoted $\overset{*}{\rightarrowtail}$ and $R^{partial}(s) = \{s' \in \mathcal{B}(Q^{\diamond} \times I) \mid s \overset{*}{\rightarrowtail} s'\}$ is the set of states reachable from state s without enabling any OR-joins (all in the context of some workflow specification).

Thus far, we excluded OR-joins from the transition relation because the OR-join implies a partial synchronisation. Whether there are enough tokens for such partial synchronisation cannot be decided locally (see Section 3.2). The required functionality is specified by Pattern 7 (Synchronising merge, [6, 70]) and can be described as follows: "an OR-join requires at least one token and waits until it is not possible to add any relevant tokens to the set of input conditions". This informal requirement is formalised in the following definition.

Definition 17. *Let $S = (Q, top, T^{\diamond}, map)$ be a specification and s_1 and s_2 two workflow states of S. $s_1 \looparrowright s_2$ if and only if $s_1 \rightarrowtail s_2$ or each of the following conditions is satisfied:*

- *There are $t \in T^{\diamond}$, $i \in I$, $c, p \in \mathcal{B}(Q^{\diamond} \times I)$ such that $join(t) = OR$, $binding_{enter}(t, i, c, p, s_1)$, and $s_2 = (s_1 - c) \uplus p$.*
- *For each $s \in R^{partial}(s_1)$, there is no $c' \in \mathcal{B}(Q^{\diamond} \times I)$ such that $binding_{enter}(t, i, c', p, s)$ and $c' > c$.*

$\looparrowright$ is the transition relation which also takes OR-joins into account. $\looparrowright$ includes all state transitions in $\rightarrowtail$ and adds transitions of type $enter$ if the number of consumed tokens cannot be increased by postponing the occurrence of the OR-join.

The reflexive transitive closure of $\looparrowright$ is denoted $\overset{*}{\looparrowright}$ and $R(s) = \{s' \in \mathcal{B}(Q^{\diamond} \times I) \mid s \overset{*}{\looparrowright} s'\}$ is the set of states reachable from state s. If ambiguity is possible, we will add subscripts, i.e., $\looparrowright_S$, $\overset{*}{\looparrowright}_S$, and R_S.

The state space $\mathcal{B}(Q^{\diamond}_S \times I)$ and transition relation $\looparrowright_S$ define a *transition system* $(\mathcal{B}(Q^{\diamond}_S \times I), \looparrowright_S)$ for S. It is also possible to define a *labelled* transition system by taking the bindings into account. For example, label a transition corresponding to $binding_{start}(t, i, c, p, s)$ as $start_{t,i}$. Such a labelled transition system can be augmented with different notions of equivalence, e.g., branching bisimilarity [24]. It is also possible to abstract from certain actions by renaming them to τ (silent action). For example, it may be useful to abstract from $exit_t$ in the semantics. The choice of a good

equivalence relation for workflow management is a topic in itself [36]. Therefore, we restrict ourselves to giving the transition system $(\mathcal{B}(Q_S^\diamond \times I), \looparrowright_S)$.

Interesting initial states for $(\mathcal{B}(Q_S^\diamond \times I), \looparrowright_S)$ are $[(\mathbf{i}_{top}, i) \mid i \in X]$ where $X \subseteq I$ such that for all $i, j \in X$: $children^*(i) \cap children^*(j) = \emptyset$ if $i \neq j$. These initial states correspond to states with a number of cases marking the initial condition for the top-level workflow. The requirement on X is needed to avoid interacting cases, i.e., tokens of different cases should not get mixed.

4.4 Soundness

Definition 17 specifies the semantics of any workflow specification as defined in Definition 2. However, some workflows may be less desirable. For example, it is possible to specify workflows that may deadlock, are unable to terminate, or have dead parts. Therefore, we define a notion of *soundness*. For the definition of soundness we assume an initial state with one case marking the initial condition of the top-level workflow.

Definition 18. *Let $S = (Q, top, T^\diamond, map)$ be a specification with initial state $[(\mathbf{i}_{top}, i)]$ for some $i \in I$.*

S has the option to complete *iff for any state $s \in R([(\mathbf{i}_{top}, i)])$: $[(\mathbf{o}_{top}, i)] \in R(s)$.*

S has no dead tasks *iff for any $t \in T$ there is a state $s \in R([(\mathbf{i}_{top}, i)])$ such that $exec_t \in q(s)$.*

S has proper completion *iff for any state $s \in R([(\mathbf{i}_{top}, i)])$: if $s \geq [(\mathbf{o}_{top}, i)]$, then $s = [(\mathbf{o}_{top}, i)]$.*

S is sound *iff S has the option to complete, has no dead tasks, and has proper completion.*

The definition of the option to complete, absence of dead tasks, proper completion, and soundness are straightforward translations of the properties given in [2, 5] for WF-nets to workflow specifications. The four properties can be extended to initial states with n cases in the initial state. However, for interesting initial states this makes no difference. Let $s_X = [(\mathbf{i}_{top}, i) \mid i \in X]$ where $X \subseteq I$ such that $X \neq \emptyset$ and for all $i, j \in X$: $children^*(i) \cap children^*(j) = \emptyset$ if $i \neq j$. S has the option to complete iff for any state $s \in R(s_X)$: $[(\mathbf{o}_{top}, j) \mid j \in X] \in R(s)$. S has no dead tasks iff for any $t \in T$ there is a state $s \in R(s_X)$ such that $exec_t \in q(s)$. S has proper completion iff for any state $s \in R(s_X)$: if $s \geq [(\mathbf{o}_{top}, j) \mid j \in X]$, then $s = [(\mathbf{o}_{top}, j) \mid j \in X]$. One could also define a notion of n-soundness where $n = size(X)$. However, a workflow specification S is sound if and only if it is n-sound. This property can be verified using the argument that tokens of different cases cannot get mixed when starting in s_X.

The notion of soundness for workflow specification can also be used to define soundness of EWF-nets.

Definition 19. *Let $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF-net. $S = (\{N\}, N, T, \emptyset)$ is the corresponding workflow specification consisting of just this EWF-net. N has the option to complete, has no dead tasks, and has proper completion if and only if S has the option to complete, has no dead tasks, and has proper completion respectively. N is sound if and only if S is sound.*

In this section we have defined the YAWL language and its semantics. In Section 4.1 we defined the syntax of a workflow specification in mathematical terms. In Section 4.3 we defined the semantics of a workflow specification by providing a transition system. Finally, in Section 4.4 we defined soundness for both workflow specifications and EWF-nets as the basic notion of correctness. Although YAWL is based on Petri nets, the language really extends (coloured) Petri nets as was shown in this section. Using YAWL it is easy, unlike Petri nets, to deal with patterns involving multiple instances, advanced synchronisation patterns, and cancellation patterns as was shown in this section.

5 Analysis

YAWL is more suitable than (high-level) Petri nets in the sense that there is direct support for several patterns that are difficult to deal with using (coloured) Petri nets. Petri nets are known for the wide variety of available analysis techniques and tools. Therefore, it is interesting to see which results can be transferred from Petri nets to YAWL. In this section, we will not address this question in detail. Instead, we give an interesting compositionality result.

A workflow specification S is composed of a set of EWF-nets Q and soundness has been defined for both workflow specifications and EWF-nets. Therefore, it is interesting to investigate whether soundness of each EWF-net in Q implies soundness of the workflow specification S. The following theorem shows that this is indeed the case.

Theorem 1. *Let $S = (Q, top, T^\diamond, map)$ be a specification. S is sound if each $N \in Q$ is sound.*

Proof. Let each $N \in Q$ be sound. We need to prove that S is sound by showing that S has the option to complete, has no dead tasks, and has proper completion. To do this we use the fact that relation $H = \{(N_1, N_2) \in Q \times Q \mid \exists_{t \in dom(map_{N_1})} map_{N_1}(t) = N_2\}$ is a tree. For each $N \in Q$: $N{\downarrow} = \{N' \in Q \mid (N, N') \in H\}$, i.e., $N{\downarrow}$ is the set of EWF-nets having N as a direct parent node in the tree structure. $N{\downarrow}^* = \{N' \in Q \mid (N, N') \in H^*\}$ is the transitive variant, i.e., all descendents of N including N.

For each $N \in Q$, we define S_N as the specification composed of EWF-nets $N{\downarrow}^*$ and top element N. It is easy to see that is a specification satisfying all the requirements stated in Definition 2. Using induction we will show that S_N is sound.

First we consider all leaf nodes. For each leaf node N: S_N is sound because $S = (\{N\}, N, T, \emptyset)$ and N is sound (cf. Definition 19).

Next we consider all nodes whose direct descendents have been shown to be sound, i.e., consider nodes N such that for each $N' \in N{\downarrow}$: $S_{N'}$ is proven to be sound. In the remainder of this proof we show that for such an N, S_N is sound. S_N has the *option to complete* because the only way to block the top level net is through blocking a $complete_t$ transition for some composite task t mapped onto some EWF-net N'. However, this is not possible because $S_{N'}$ is sound. Moreover, there cannot be any tokens left in $S_{N'}$ because the moment a token is put onto $\mathbf{o}_{N'}$, $S_{N'}$ has no other tokens corresponding to the same instance. S_N has *no dead tasks* because each task t in N can be enabled. As a result any of the tasks in $S_{N'}$ can be activated if t mapped onto some EWF-net N'. S_N has *proper completion* for similar reasons as it has the option to

complete. As a result S_N is *sound* because S_N has the option to complete, has no dead tasks, and has proper completion.

This can be repeated and using induction we can prove that $S_{top} = S$ is sound. □

This theorem shows that the correctness of a YAWL workflow specification can verified in a compositional way. This result has been included in this paper to demonstrate that the formal semantics of YAWL allow for the development of analysis methods and supporting theories. The goal is to transfer as many results from the Petri-net domain to YAWL as possible. Given the expressiveness of YAWL, these results are directly applicable to a range of existing workflow languages. For example, we expect that for restricted YAWL specifications we can use invariants comparable to place and transition invariants in Petri nets [12, 34, 49].

6 Related work

In this section some workflow languages used in the literature will be analysed in terms of their support for the patterns as presented in [6]. The aim is to provide some insight into the relative strength of YAWL with respect to these approaches in terms of support for the control flow perspective. The results are summarised in Table 1. It should be noted that a pattern-based analysis of 13 commercial workflow offerings can be found in [6]. For an analysis of UML activity diagrams 1.4 in terms of (some of) the patterns, we refer to [14]. BPEL4WS, a proposed standard for web service composition, has been analysed in [69], while another such proposed standard, BPML, has been analysed in [4]. WfMC's XPDL [67] is assessed in terms of the patterns in the appendix. Note that the appendix also provides a general explanation of how to correctly interpret the various ratings.

The approach described in ADEPTflex [48] takes structured models as a starting point, hence splits and joins can be matched, and loops have a unique entry and a unique exit (hence arbitrary loops are not supported, and neither is implicit termination). In this structured context, the discriminator is supported through "parallel branching with final selection". The deferred choice is supported as decisions may be made by users in which case all possible options are offered and once one option is selected, the other options are withdrawn. One can use "soft synchronization" to capture the synchronising merge. It should be noted that the corresponding edge and the edge corresponding to "strict synchronization" can be used between different parallel branches, hence the approach goes beyond approaches that are fully structured. Apart from the deferred choice, other state-based patterns are not supported. In addition, there is no support for cancellation and multiple active instances of the same activity within the same case.

Apart from the basic control flow patterns, OPENflow [25] supports the multi-choice and multiple instances with a priori design time knowledge (note though that this pattern always receives a "+" rating if parallelism/synchronisation is supported). Interestingly, it also supports the notion of recursive decomposition: "Genesis tasks can also be utilised to specify workflow applications that contain recursive executions, that is a task structure whose execution will potentially cause the execution of its own structure as one of its sub-tasks." [25]. There is no pattern corresponding to recursive decomposition.

pattern	product									
	$\text{ADEPT}_{\text{flex}}$ [48]	OPENflow [25]	EPC [35]	Mentor [68, 45, 44]	CTR [10, 55]	Meteor [56]	Mobile [32]	WASA [65]	Exotica [43]	CFs [13]
1 (seq)	+	+	+	+	+	+	+	+	+	+
2 (par-spl)	+	+	+	+	+	+	+	+	+	+
3 (synch)	+	+	+	+	+	+	+	+	+	+
4 (ex-ch)	+	+	+	+	+	+	+	+	+	+
5 (simple-m)	+	+	+	+	+	+	+	+	+	+
6 (m-choice)	-$^{(i)}$	+	+	-	+	+	+	+	+	-
7 (sync-m)	+$^{(ii)}$	-	+	-	+	-	-	+	+	-
8 (multi-m)	-	-	-	-	-	+	-	-	-	-
9 (disc)	+$^{(ii)}$	-	-	+	-	+/-	+	-	-	-
10 (arb-c)	-	-$^{(i)}$	+	-	-	+	-	-	-	-
11 (impl-t)	-	-$^{(i)}$	+	-	-	-	-	+	+	+
12 (mi-no-s)	-	-	-	-	-	+	-	-	-	-
13 (mi-dt)	+	+	+	+	+	+	+	+	+	+
14 (mi-rt)	-	-	-	-	-	-	-	-	-$^{(iii)}$	-
15 (mi-no)	-	-	-	-	-	-	-	-	-	-
16 (def-c)	+	-	-	+	+	-	-	-	-	-
17 (int-par)	-	-	-	-	-	-	+	-	-	-
18 (milest)	-	-	-	+/-	-	-	-	-	-	-
19 (can-a)	-	-	-	+	-	-	-	-	-	-
20 (can-c)	-	-	-	+	-	-	-	-	-	-

Table 1. Main results of evaluating workflow languages used in the literature using the workflow patterns [6, 70]. (i) Support for this pattern could not be deduced from the paper. (ii) Only in a structured context. (iii) If the Bundle concept of FlowMark is used this "-" should change into a "+".

EPCs [35] allow for arbitrary cycles, implicit termination, multi-choice, and the synchronising merge, but do not support multiple active instances of the same activity within the same case, the discriminator, the multi-merge, cancellation, and the state-based patterns. One could argue that the multi-choice is not supported because the OR-split connector typically does not have a explicit specification. However, this also holds for some of the other constructs and is directly caused by the fact that it is typically used in an informal manner. As indicated in the discussion in Section 3.2 the semantics of the OR-join is also under-specified, and thus the support for the synchronising merge may also be debated.

In the Mentor project [68, 45, 44] state and activity charts are used for workflow specification. These languages were proposed in the context of the STATEMATE system described in [26]. The evaluation given here is based on this reference. Since state charts are defined in terms of classical automata, the notion of being in a state more than once at a given point in time (which would correspond to having mulitple tokens in the same place in a Petri net) is not supported. As a result, there is inherently no support for multiple active instances of the same activity within the same case, nor for the multi-merge. Also, there is no support for arbitrary cycles. Specifically, when a transition crosses the boundary of a concurrent region in a state chart, the semantics is that all the states in that region and in its sibling regions are exited if this transition is taken. Hence it is not possible to express arbitrary cycles crossing concurrent regions with the same semantics as in YAWL. State charts must have a single final state hence there is no support for implicit termination. Additionally, there are no constructs corresponding to the multi-choice, the synchronising merge, and interleaved parallel routing. Finally, the milestone pattern receives a "+/-" as in the condition part of an ECA rule one can test whether the execution is in a given state through the predicate in. It should be remarked that state charts offer an operator called "History", which allows resumption of the state of execution that existed prior to a cancellation event. This feature does not have a corresponding pattern.

Concurrent Transaction Logic [8] (CTR) is used in [10, 55] for workflow specification. Apart from the basic control flow patterns, CTR supports the multi-choice, the synchronising merge, multiple instances with a priori design time knowledge, and the deferred choice. CTR goes beyond fully structured approaches when it comes to parallel structures as messaging can be used to achieve synchronisation between parallel branches (see Example 2.2 in [8]).

The evaluations of Meteor and Mobile are taken from [6]. Note that in [6] it is remarked that Mobile is extensible and that hence the evaluation restricts itself to the standard functionality offered.

In terms of support for the workflow patterns, WASA [63, 65, 66] corresponds to MQ/Series Workflow (which is evaluated in [6]). Contrary to MQ/Series Workflow, iteration in WASA is supported through recursion: "... a workflow schema *i* can appear as a sub-workflow schema of itself ..." [65] (page 50). As stated before, there is no pattern corresponding to this feature. Note that the focus of WASA is not on expressiveness but on supporting workflow change.

The Exotica Research Project is described in [43] where it is stated that "... we have focused on a particular commercial product, *FlowMark*, IBM's workflow product

...". As remarked in [6], the *Bundle* constructs is supported by FlowMark, but not by MQSeries/Workflow version 3.1. If this feature was used in Exotica, this would mean that in Table 1, support for multiple instances with a priori runtime knowledge, would change from a "-" to a "+".

The WASA and Exotica research projects illustrate that several research prototypes have been influenced by the IBM product FlowMark and its successors MQSeries Workflow and WebSphere MQ Workflow. See [41] for more details. It is also interesting to see that FlowMark-specific concepts such as "Death Path Elimination" have been adopted in web service composition languages such as WSFL and BPEL4WS.

Communicating Flowcharts (CFs) [13] support the basic control flow patterns, implicit termination, and multiple instances with a priori design time knowledge, but none of the other workflow patterns.

The Vortex approach described in [30] does not really amend itself to a pattern-based analysis, as all control flow dependencies are captured through the data perspective.

A lot of work has been done in the area of transactional aspects of workflows, see e.g. [21, 28, 62, 64]. It seems that transactional aspects play a role at a different level than typical control flow aspects, though the separation between the two is not always clear-cut (consider e.g. the cancellation patterns). The relation between YAWL and transactional concepts needs to be researched.

7 Conclusion

Through the analysis of a number of languages supported by workflow management systems a number of patterns was distilled in previous work. While all these patterns can be realised in high-level Petri nets, some of these patterns can only be realised in a rather indirect way requiring a lot of specification effort. The language YAWL was designed, based on Petri nets as to preserve their strengths for the specification of control-flow dependencies in workflows, with extensions that allowed for straightforward specification of these patterns.

YAWL can be considered a very powerful workflow language, built upon experiences with languages supported by contemporary workflow management systems. While not a commercial language itself it encompasses these languages, and, in addition, has a formal semantics. Such an approach is in contrast with e.g. WfMC's XPDL [67] which takes commonalities between various languages as a starting point and does not have formal semantics. Its design hopefully allows YAWL to be used for the purposes of the study of expressiveness and interoperability issues.

The definition of YAWL presented in this paper only supports the control-flow perspective. However, it is important that relations with the other perspectives relevant in the context of workflow, e.g., the data and the resource perspectives, are investigated and formalised. Therefore, YAWL has been extended with these perspectives. For YAWL to be more applicable in the area of web services and Enterprise Application Integration it is also desirable that support for communication patterns (e.g. the ones specified in [52]) is built-in. Another direction for future research is to extend YAWL with transaction capabilities.

Besides extending YAWL for other perspectives, we are also working on analysis techniques and tool support. We are building a design tool and engine for YAWL. At this point in time the engine of YAWL fully supports the language proposed in this paper.

Acknowledgments. The authors would like to thank Alistair Barros, Marlon Dumas, Bartek Kiepuszewski, and Petia Wohed for their collaborative work on the workflow patterns. We would also like to thank Marlon Dumas for his remarks in relation to the evaluation of state charts. Moreover, we would particularly like to thank Lachlan Aldred for implementing the YAWL engine and commenting on earlier versions of the paper. Finally, we would like to thank the reviewers for their valuable suggestions.

Disclaimer. We, the authors and the associated institutions, assume no legal liability or responsibility for the accuracy and completeness of any information about workflow products, workflow languages, and standards contained in this paper. However, we made all possible efforts to ensure that the results presented are, to the best of our knowledge, up-to-date and correct.

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, M. Dumas, A.H.M. ter Hofstede, and P. Wohed. Pattern-Based Analysis of BPML (and WSCI). QUT Technical report, FIT-TR-2002-05, Queensland University of Technology, Brisbane, Australia, 2002. Available via www.citi.qut.edu.au/pubs/technical/pattern_based_analysis_BPML.pdf.
5. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
6. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
7. Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
8. A.J. Bonner and M. Kifer. Concurrency and Communication in Transaction Logic. In M.J. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming, September 2-6, 1996, Bonn, Germany*, pages 142–156. MIT Press, 1996.
9. P. Chrząstowski-Wachtel, B. Benatallah, R. Hamadi, M. O'Dell, and A. Susanto. A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 336–353. Springer-Verlag, Berlin, 2003.
10. H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic Based Modeling and Analysis of Workflows (Extended Abstract). In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 25–33. ACM Press, 1998.

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. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
13. G. Dong, R. Hull, B. Kumar, J. Su, and G. Zhou. A Framework for Optimizing Distributed Workflow Executions. In R. Connor and A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming: 7th International Workshop on Database Programming Languages, DBPL'99, Kinloch Rannoch, UK, September 1999. Revised Papers*, volume 1949 of *Lecture Notes in Computer Science*, pages 152–167, Heidelberg, Germany, 2000. Springer-Verlag.
14. M. Dumas and A.H.M. ter Hofstede. UML activity diagrams as a workflow specification language. In M. Gogolla and C. Kobryn, editors, *Proc. of the 4th Int. Conference on the Unified Modeling Language (UML01)*, volume 2185 of *LNCS*, pages 76–90, Toronto, Canada, October 2001. Springer Verlag.
15. 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.
16. FileNet. *Visual WorkFlo Design Guide*. FileNet Corporation, Costa Mesa, CA, USA, 1997.
17. L. Fischer, editor. *Workflow Handbook 2001, Workflow Management Coalition*. Future Strategies, Lighthouse Point, Florida, 2001.
18. Forté. *Forté Conductor Process Development Guide*. Forté Software, Inc, Oakland, CA, USA, 1998.
19. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, Massachusetts, 1997.
20. Fujitsu. *i-Flow Developers Guide*. Fujitsu Software Corporation, San Jose, CA, USA, 1999.
21. G. Alonso and D. Agrawal and A. El Abbadi and M. Kamath and R. Günthör and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*. IEEE Computer Society, 1996.
22. 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.
23. H. J. Genrich and P. S. Thiagarajan. A Theory of Bipolar Synchronization Schemes. *Theoretical Computer Science*, 30(3):241–318, 1984.
24. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.
25. J.J. Halliday, S.K. Shrivastava, and S.M. Wheater. Flexible Workflow Management in the OPENflow System. In *4th International Enterprise Distributed Object Computing Conference (EDOC 2001), 4-7 September 2001, Seattle, Washington, Proceedings*, pages 82–92. IEEE Computer Society, 2001.
26. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.
27. K.M. van Hee. *Information System Engineering: a Formal Approach*. Cambridge University Press, 1994.
28. I. Houston, M.C. Little, I. Robinson, S.K. Shrivastava, and S.M. Wheater. The CORBA Activity Service Framework for Supporting Extended Transactions. In R. Guerraoui, editor, *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms,*

Heidelberg, Germany, November 12-16, 2001. Proceedings, volume 2218 of *Lecture Notes in Computer Science*, pages 197–215. Springer-Verlag, 2001.
29. HP. *HP Changengine Process Design Guide*. Hewlett-Packard Company, Palo Alto, CA, USA, 2000.
30. R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative Workflows that Support Easy Modification and Dynamic Browsing. In G. Georgakopoulos, W. Prinz, and A.L. Wolf, editors, *Work Activities Coordination and Collaboration (WACC'99)*, pages 69–78, San Francisco, February 1999. ACM press.
31. IBM. *IBM MQSeries Workflow - Getting Started With Buildtime*. IBM Deutschland Entwicklung GmbH, Boeblingen, Germany, 1999.
32. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
33. 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.
34. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
35. 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.
36. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003. Available via http://www.tm.tue.nl/it/research/patterns.
37. 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.
38. B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In B. Wangler and L. Bergman, editors, *Proc. of the 12th Int. Conference on Advanced Information Systems Engineering (CAiSE00)*, volume 1789 of *LNCS*, pages 431–445, Stockholm, Sweden, June 2000. Springer Verlag.
39. 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.
40. P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997.
41. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
42. 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.
43. C. Mohan, G. Alonso, R. Günthör, and M. Kamath. Exotica: A Research Perspective on Workflow Management Systems. *IEEE Data Engineering Bulletin*, 18(1):19–26, 1995.
44. P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Integrating Light-Weight Workflow Management Systems within Existing Business Environments. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Australia*, pages 286–293. IEEE Computer Society, 1999.
45. P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Workflow History Management in Virtual Enterprises Using a Light-Weight Workflow Management System. In *Proceedings of the Ninth International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises, 23-24 March, 1999, Sydney, Australia*, pages 148–155, 1999.

46. 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.
47. C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Fakultät für Mathematik und Physik, Technische Hochschule Darmstadt, Darmstadt, Germany, 1962.
48. M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
49. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
50. D. Riehle and H. Züllighoven. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.
51. P. Rittgen. Modified EPCs and their Formal Semantics. Technical report 99/19, University of Koblenz-Landau, Koblenz, Germany, 1999.
52. W.A. Ruh, F.X. Maginnis, and W.J. Brown. *Enterprise Application Integration: A Wiley Tech Brief*. John Wiley and Sons, New York, 2001.
53. F. Rump. *Geschäftsprozessmanagement auf der Basis ereignisgesteuerter Prozessketten*. Reihe Wirtschaftsinformatik, Teubner Verlag, Germany, 1999.
54. SAP. *WF SAP Business Workflow*. SAP AG, Walldorf, Germany, 1997.
55. P. Senkul, M. Kifer, and I.H. Toroslu. A Logical Framework for Scheduling Workflows under Resource Allocation Constraints. In *Proceedings of the 28th International Conference on Very Large Data Bases, Hong Kong, China, 2002*, pages 694–705, 2002.
56. 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.
57. Eastman Software. *RouteBuilder Tool User's Guide*. Eastman Software, Inc, Billerica, MA, USA, 1998.
58. Software-Ley. *COSA 3.0 User Manual*. Software-Ley GmbH, Pullheim, Germany, 1999.
59. Staffware. *Staffware 2000 / GWD User Manual*. Staffware plc, Berkshire, United Kingdom, 2000.
60. Tibco. *TIB/InConcert Process Designer User's Guide*. Tibco Software Inc., Palo Alto, CA, USA, 2000.
61. Verve. *Verve Component Workflow Engine Concepts*. Verve, Inc., San Francisco, CA, USA, 2000.
62. G. Vossen. Transactional Workflows (tutorial). In F. Bry, R. Ramakrishnan, and K. Ramamohanarao, editors, *Proceedings of the 5th International Conference on Deductive and Object-Oriented Databases (DOOD'97)*, volume 1341 of *Lecture Notes in Computer Science*, pages 20–25. Springer-Verlag, Berlin, 1997.
63. G. Vossen and M. Weske. The WASA2 Object-Oriented Workflow Management System. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 587–589. ACM Press, 1999.
64. G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, San Francisco, CA, 2002.
65. M. Weske. Formal Foundation, Conceptual Design, and Prototypical Implementation of Workflow Management Systems. Habilitation's thesis, University of Münster, Germany, 2000.
66. M. Weske. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In R. Sprague, editor, *Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-34)*. IEEE Computer Society Press, Los Alamitos, California, 2001.

67. WFMC. Workflow Management Coalition Workflow Standard: Workflow Process Definition Interface – XML Process Definition Language (XPDL) (WFMC-TC-1025). Technical report, Workflow Management Coalition, Lighthouse Point, Florida, USA, 2002.

68. D. Wodtke, J. Weissenfels, G. Weikum, and A.K. Dittrich. The Mentor Project: Steps Toward Enterprise-Wide Workflow Management. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*. IEEE Computer Society, 1996.

69. 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, and T.W. Ling, editors, *22nd International Conference on Conceptual Modeling (ER 2003)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2003. (to appear).

70. 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 and/or YAWL offer direct support (indicated by a "+"), partial direct support (indicated by a "+/-"), or no direct support (indicated by a "-"). For comparison, we also included the WfMC's XPDL [67]. It is important to correctly interpret the "+", "+/-", and "-". A "+" is only given if the language offers a feature that allows for a direct realisation of the pattern without any restrictions, without the need for coding inside or outside the tool, and without the need to unfold the construct into other constructs. If the language offers a feature that allows for a direct realisation of the pattern but also imposes restrictions, it is rated with "+/-". If the language only offers constructs to realise the pattern indirectly (through coding, unfolding using other constructs, etc.) or does not support the construct at all, it is rated "-". For more details on the rating on the basis of patterns we refer to [6] where 15 workflow products are rated using the same principle.

pattern	XPDL	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 synchronising merge is not supported because the designer has to keep track of the number of parallel threads and decide to merge or synchronise 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 synchronising 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 synchronisation 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).