Design and implementation of the YAWL system

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

[1] Department of Technology Management, Eindhoven University of Technology, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl
[2] Centre for IT Innovation, Queensland University of Technology, Australia
{l.aldred,m.dumas,a.terhofstede}@qut.edu.au

Abstract. This paper describes the implementation of a system supporting YAWL (Yet Another Workflow Language). YAWL is based on a rigorous analysis of existing workflow management systems and related standards using a comprehensive set of workflow patterns. This analysis shows that contemporary workflow systems, relevant standards (e.g. XPDL, BPML, BPEL4WS), but also theoretical models such as Petri nets have problems supporting essential patterns. This inspired the development of YAWL by taking Petri nets as a starting point and introducing mechanisms that provide direct support for the workflow patterns identified. As a proof of concept we have developed a workflow management system supporting YAWL. In this paper, we present the architecture and functionality of the system and zoom into the control-flow, data, and operational perspectives.

1 Introduction

In the area of workflow one is confronted with a plethora of products (commercial, free and open source) supporting languages that differ significantly in terms of concepts, constructs, and their semantics. One of the contributing factors to this problem is the lack of a commonly agreed upon formal foundation for workflow languages. Standardization efforts, e.g. XPDL [20] proposed by the WfMC, have essentially failed to gain universal acceptance and have not in any case provided such a formal basis for workflow specification. The lack of well-grounded standards in this area has induced several issues, including minimal support for migration of workflow specifications, potential for errors in specifications due to ambiguities, and lack of a reference framework for comparing the relative expressive power of different languages (though some work in this area is reported in [13]).

The workflow patterns initiative [4] aims at establishing a more structured approach to the issue of the specification of control flow dependencies in workflow languages. Based on an analysis of existing workflow management systems and applications, this initiative identified a collection of patterns corresponding to typical control flow dependencies encountered in workflow specifications, and documented ways of capturing these dependencies in existing workflow languages. These patterns have been used as a benchmark for comparing process definition languages and in tendering processes for evaluating workflow offerings. See `www.workflowpatterns.com` for extensive documentation, flash animations of each pattern, and evaluations of standards and systems.

While workfbw patterns provide a pragmatic approach to control fbw specification in workfbws, Petri nets provide a more theoretical approach. Petri nets [17] form a model for concurrency with a formal foundation, an associated graphical representation, and a collection of analysis techniques. These features, together with their direct support for the notion of state (required in some of the workfbw patterns), makes them attractive as a foundation for control fbw specification in workfbws. However, even though Petri nets (including higher-order Petri nets such as Colored Petri nets [12]) support a number of the identified patterns, they do not provide direct support for the cancellation patterns (in particular the cancellation of a whole case or a region), the synchronizing merge pattern (where all active threads need to be merged, and branches which cannot become active need to be ignored), and patterns dealing with multiple active instances of the same activity in the same case [2]. This realization motivated the development of YAWL [3] (Yet Another Workfbw Language) which combines the insights gained from the workfbw patterns with the benefits of Petri nets. It should be noted though that YAWL is not simply a set of macros defined on top of Petri nets. Its semantics is not defined in terms of Petri nets but rather in terms of a transition system.

As a language for the specification of control fbw in workfbws, YAWL has the benefit of being highly expressive and suitable, in the sense that it provides direct support for all the workfbw patterns, while the reviewed workfbw languages provide direct support for only a subset of them. In addition, YAWL has a formal semantics and offers graphical representations for many of its concepts. The expressive power and formal semantics of YAWL make it an attractive candidate to be used as an intermediate language to support translations of workfbws specified in different languages.

When YAWL was first proposed no implementation was available. Recently, implementation efforts have resulted in a first version of a prototype supporting YAWL. With respect to the various perspectives from which workfbws can be considered (e.g. control-fbw, data, resource, and operational [11]), YAWL initially focused exclusively on the control fbw perspective. Subsequently, a novel approach for dealing with the data perspective has been designed and incorporated into the prototype. In addition, an approach has been designed (although not yet implemented) to deal with the operational perspective on the basis of a service-oriented architecture.

This paper discusses salient aspects and issues related to the design and implementation of the YAWL system, including the proposed extensions for dealing with the data and operational perspectives. In short, the main contributions of the paper are:

– A discussion of the implementation of the control fbw perspective of YAWL;
– A discussion of the data perspective of YAWL and its implementation;
– A discussion of a proposal for the incorporation of the operational perspective into YAWL through the use of a service-oriented architecture.

The remainder of the paper is organized as follows. After a brief overview of related work we introduce the YAWL language. Section 4 describes the architecture of the YAWL system and motivates design decisions. Section 5 discusses the control-fbw, data, and operational perspectives in more detail. Section 6 briefly discusses an example. Section 7 concludes the paper.

2 Related work

YAWL [3] is based on a line of research grounded in Petri net theory [1, 13] and the 20 workflow patterns documented in [4]. In previous publications we have evaluated contemporary systems, languages, and standards using these patterns. An analysis of 13 commercial workflow offerings can be found in [4] while an analysis of 10 workflow languages proposed by the academic community is described in [3]. Commercial systems that have been evaluated include Ley COSA , Filenet's Visual Workflow, SUN's Forté Conductor, Lotus Domino Workflow, IBM MQSeries/Workflow, Staffware, Verve Workflow, Fujitsu's I-Flow, TIBCO InConcert, HP Changengine, SAP R/3 Workflow, Eastman, and FLOWer. Examples of academic prototypes that have been evaluated using the patterns are Meteor, Mobile [11], ADEPTflex [18], OPENflow, Mentor [21], and WASA [19]. For an analysis of UML activity diagrams in terms of (some of) the patterns, we refer to [8]. BPEL4WS, a proposed standard for web service composition, has been analyzed in [22]. Analyses of BPMI's BPML [5] and WfMC's XPDL [20] using the patterns are also available via `www.workflowpatterns.com`. In total, more than 30 languages/systems have been evaluated and these evaluations have driven the development of the YAWL language. Given that this paper focuses on the *design and implementation* of the YAWL system, we will not discuss these previous evaluations, referring the reader to the above citations.

As an open source workflow system, YAWL joins the ranks of a significant number of previous initiatives: 18 open source workflow systems are reported in [16]. Again, the distinctive feature of YAWL with respect to these systems is in the combination of its expressive power, formal foundation, and support for graphical design, complemented by its novel approach to deal with the data and the operational perspective of workflow by leveraging emerging XML and Web services technologies.

3 YAWL language

Before describing the architecture and implementation of the YAWL system, we introduce the distinguishing features of YAWL. As indicated in the introduction, YAWL is based on Petri nets. However, to overcome the limitations of Petri nets, YAWL has been extended with features to facilitate patterns involving multiple instances, advanced synchronization patterns, and cancellation patterns. Moreover, YAWL allows for hierarchical decomposition and handles arbitrarily complex data.

Figure 1 shows the modeling elements of YAWL. At the syntactic level, YAWL extends the class of workflow nets described in [1] with multiple instances, composite tasks, OR-joins, removal of tokens, and directly connected transitions. YAWL is thus 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 its own semantics and specifically designed for workflow specification.

A *workflow specification* in YAWL is a set of *process definitions* which form a hierarchy. *Tasks*[1] are either *atomic tasks* or *composite tasks*. Each composite task refers

[1] We use the term *task* rather than *activity* to remain consistent with earlier work on workflow nets [1].

to a process definition at a lower level in the hierarchy (also referred to as its decomposition). Atomic tasks form the leaves of the graph-like structure. There is one process definition without a composite task referring to it. This process definition is named the *top level workflow* and forms the root of the graph-like structure representing the hierarchy of process definitions.

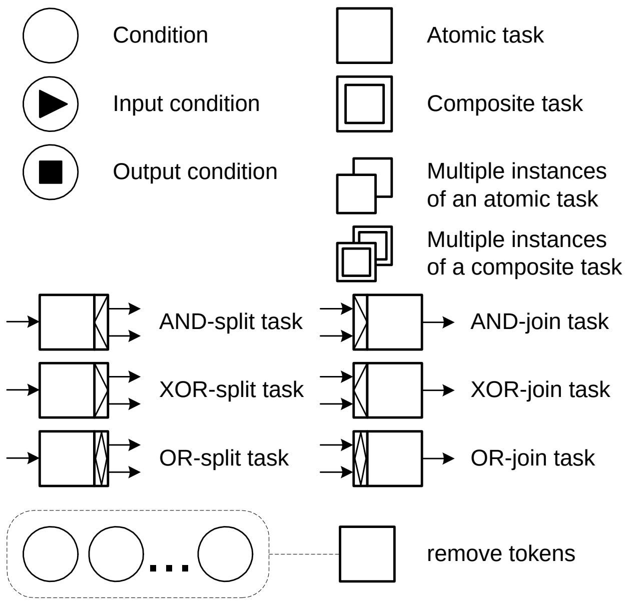
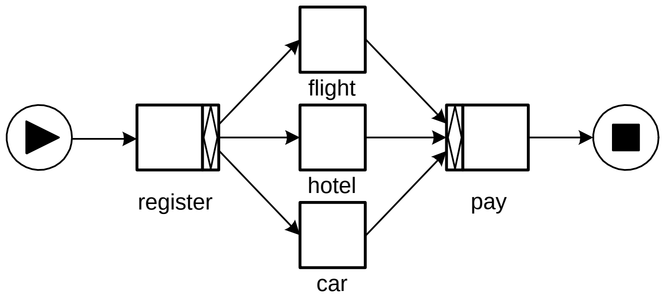


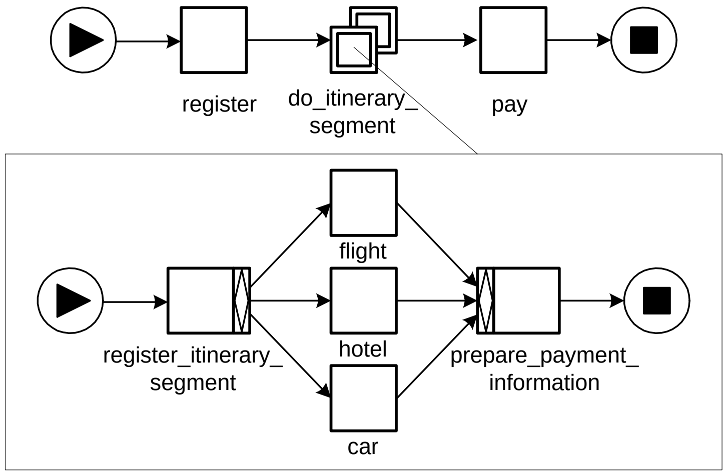
Fig. 1. Symbols used in YAWL.

Each process definition consists of *tasks* (whether composite or atomic) and *conditions* which can be interpreted as places. Each process definition has one unique *input condition* and one unique *output condition* (see Figure 1). 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.

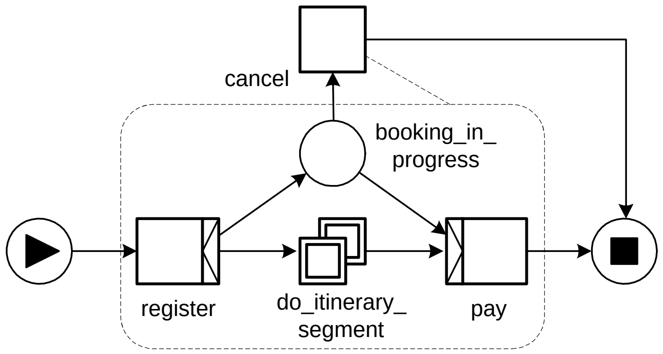
Each task (either composite or atomic) can have multiple instances as indicated in Figure 1. We adopt the notation described in [1] for AND/XOR-splits/joins as also shown in Figure 1. Moreover, we introduce OR-splits and OR-joins corresponding respectively to Pattern 6 (Multi choice) and Pattern 7 (Synchronizing merge) defined in [4]. Finally, Figure 1 shows that YAWL provides a notation for removing tokens from a specified region denoted by dashed rounded rectangles and lines. The enabling of the task that will perform the cancellation may or may not depend on the tokens within the region to be “canceled”. In any case, the moment this task executes, all tokens in this region are removed. This notation allows for various cancellation patterns.



(a) After registering the request for a trip, a flight, a hotel, and/or a car are booked followed by a payment.



(b) A trip may consist of several legs. The sub-process is instantiated for each leg.



(c) Again the sub-process is instantiated for each leg but now it is possible to cancel the whole trip by removing tokens from the region indicated.

Fig. 2. Three YAWL specifications.

To illustrate YAWL we use the three examples shown in Figure 2. The first example (a) illustrates that YAWL allows for the modeling of advanced synchronization patterns. Task *register* is an 'OR-split' (Pattern 6: Multi-choice) and task *pay* is an 'OR-join' (Pattern 7: Synchronizing merge). This implies that every registration step is followed by a set of booking tasks *flight*, *hotel*, and/or *car*. It is possible that all three booking tasks are executed but it is also possible that only one or two booking tasks are executed. The YAWL OR-join synchronizes only if necessary, i.e., it will synchronize only the booking tasks that were actually selected. Note that the majority of systems do not support the Synchronizing merge (i.e., Pattern 7). A few systems support it (e.g., IBM's MQSeries Workflow, Lotus Domino Workflow, and Eastman Workflow) but restrict its application. For example, in order to simplify the implementation of the OR-join, MQSeries Workflow [10] does not support loops. [2].

Figure 2(a) does not show the data aspect. The YAWL specification for this example has 10 variables ranging from the name of the customer to flight details. For the routing of the case there are three Boolean variables *want_flight*, *want_hotel*, and *want_car* to select which of the booking tasks need to be executed.

Figure 2(b) illustrates another YAWL specification. In contrast to the first example a trip may include multiple stops, i.e., an itinerary may include multiple segments. For example, a trip may go from Amsterdam to Singapore, from Singapore to Brisbane, from Brisbane to Los Angeles, and finally from Los Angeles to Amsterdam and thus entail four itinerary segments. Each segment may include a flight (most likely) but may also include a hotel booking or a car booking (at the destination). Figure 2(b) shows that multiple segments are modeled by multiple instances of the composite task *do_itinerary_segment*. This composite task is linked to the process definition also shown in Figure 2(b). In the case of multiple instances it is possible to specify upper and lower bounds for the number of instances. It is also possible to specify a threshold for completion that is lower than the actual number of instances, i.e., the construct completes before all of its instances complete. In the example at hand this does not make sense since each segment must be booked. Another setting is available to indicate whether an instance can be added while executing other instances. In this example this would mean that while booking segments, a new segment is defined and added to the itinerary. In the specification corresponding to Figure 2(b) we assume the multiple instances to be 'static', i.e., after completing task *register* the number of instances is fixed. Again, the diagram does not show the data aspect. There are similar variables as in the first example. However, a complicating factor is that each of the instances will use private data which needs to be aggregated. We return to this in Section 5.2. For the moment, it suffices to see that YAWL indeed supports the patterns dealing with multiple instances (Patterns 12-15). Note that only few systems support multiple instances. FLOWer [6] is one of the few systems directly supporting multiple instances.

Finally we consider the YAWL specification illustrated in Figure 2(c). Again composite task *do_itinerary_segment* is decomposed into the process definition shown in Figure 2(b). Now it is however possible to withdraw bookings by executing task *cancel*. Task *cancel* is enabled if there is a token in *booking_in_progress*. If the environment decides to execute cancel, everything inside the region indicated by the dashed rectan-

[2] Instead of loops, MQSeries Workflow supports blocks with exit condition.

gle will be removed. In this way, YAWL provides direct support for the cancellation patterns (Patterns 19 and 20). Note that support for these patterns is typically missing or very limited in existing systems (e.g. Staffware has a cancellation concept but a cancellation can only refer to a single task and not to an arbitrary set of tasks).

In this section we illustrated some of the features of the YAWL language. The language has an XML syntax and is specified in terms of an XML schema. See `www.citi.qut.edu.au/yawl/` for the XML syntax of the language. In Section 5 we will show some fragments of the language. However, before going into detail, we first present the 'bigger picture' by describing the YAWL architecture.

4 YAWL architecture

To support the YAWL language introduced in the previous section, we have developed a system using state-of-the-art technology. In this section, we describe the overall architecture of this system, which is depicted in Figure 3. Workfbw specifications are designed using the *YAWL designer* and deployed into the *YAWL engine* which, after performing all necessary verifications and task registrations, stores these specifications in the *YAWL repository*, which manages a collection of "runable" workfbw specifications. Once successfully deployed, workfbw specifications can be instantiated through The YAWL engine, leading to workfbw instances (also called cases). The engine handles the execution of these cases, i.e. based on the state of a case and its specification, the engine determines which events it should offer to the environment.

The environment of a YAWL system is composed of so-called *YAWL services*. Inspired by the "web services" paradigm, end-users, applications, and organizations are all abstracted as services in YAWL. Figure 3 shows four YAWL services: (1) *YAWL worklist handler*, (2) *YAWL web services broker*, (3) *YAWL interoperability broker*, and (4) *custom YAWL services*. The YAWL worklist handler corresponds to the classical worklist handler (also named "inbox") present in most workfbw management systems. It is the component used to assign work to users of the system. Through the worklist handler users can accept work items and signal their completion. In traditional workfbw systems, the worklist handler is embedded in the workfbw engine. In YAWL however, it is considered to be a service decoupled from the engine. The YAWL web services broker is the glue between the engine and other web services. Note that it is unlikely that web services will be able to directly connect to the YAWL engine, since they will typically be designed for more general purposes than just interacting with a workfbw engine. Similarly, it is desirable not to adapt the interface of the engine to suit specific services, otherwise, this interface will need to cater for an undetermined number of message types. Accordingly, the YAWL web services broker acts as a mediator between the YAWL engine and external web services that may be invoked by the engine to delegate tasks (e.g. delegating a "payment" task to an online payment service). The YAWL interoperability broker is a service designed to interconnect different workfbw engines. For example, a task in one system could be subcontracted to another system where the task corresponds to a whole process. To illustrate that there is not a fixed set of YAWL services we included a custom YAWL service. A custom service connects the engine with an entity in the environment of the system. For example a custom YAWL

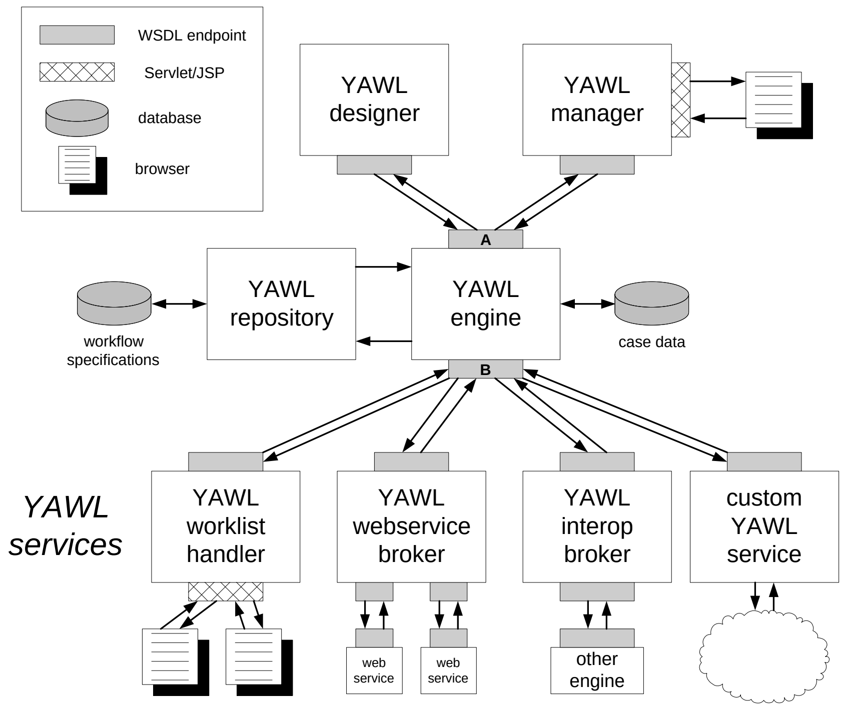


Fig. 3. YAWL architecture.

service could offer communication with mobile phones, printers, assembly robots, etc. Note that it is also possible that there are multiple services of the same type, e.g. multiple worklist handlers, web services brokers, and interoperability brokers. For example, there may exist multiple implementations of worklist handlers (e.g., customized for a specific application domain or organization) and the same worklist handler may be instantiated multiple times (e.g., one worklist handler per geographical region).

Workflow specifications are managed by the YAWL repository and workflow instances (i.e. cases) are managed by the YAWL engine. Clearly, there is also a need for a management tool that can be used to control workflow instances manually (e.g. deleting a workflow instance or a workflow specification), providing information about the state of running workflow instances, and details or aggregated data about completed instances. This is the role of the *YAWL manager*.

Figure 3 also shows the various interfaces of YAWL. The YAWL engine has two groups of interfaces: (A) interfaces capturing the interactions between the YAWL designer and the YAWL manager on the one hand, and the YAWL engine on the other; and (B) interfaces capturing the interactions between the YAWL services and the YAWL engine. The group of interfaces (A) corresponds to Interface 1 (Process Definition tools) and Interface 5 (Administration and Monitoring tools) of the reference model of the Workflow Management Coalition (WfMC) [15, 9]. The group set of interfaces (B) corresponds to WfMC's Interface 2-3 (Workflow Client Applications and Invoked Applications), and Interface 4 (Workfbw Interoperability). Both interfaces (A and B) are specified in WSDL. Users interact with the YAWL system through a Web browser, i.e. both the YAWL manager and the YAWL worklist handler offer HTML front-ends.

When considering the YAWL architecture there is one fundamental difference with respect to existing workfbw management systems: The YAWL engine deals with control-fbw and data but not explicitly with users. In addition, the engine abstracts from differences between users, applications, organizations, etc. To achieve this, the YAWL system leverages principles from service-oriented architectures: external entities either offer services or require services. In a traditional workfbw management system, the engine takes care of the 'What', 'When', 'How', and 'By whom'. In YAWL, the engine takes care of the 'What' and 'When' while the YAWL services take care of the 'How' and 'By whom'. By separating these concerns it is possible to implement a highly efficient engine while allowing for customized functionality. For example, it is possible to build worklist handlers supporting specific organizations or domains, e.g., processes where a team of professionals work on the same activity at the same time. It should be noted that the architecture of YAWL is similar to the architecture envisioned in the context of web service composition languages like BPEL4WS, WSCI, BPML, etc. However, these languages typically only consider control routing and data transformations, while YAWL is not limited to these aspects and also addresses issues such as work distribution and management.

Implementation notes. Although the current implementation of YAWL is complete in the sense that it is able to run workfbw instances, it does not provide all the components and functionality described in Figure 3. The YAWL engine is fully implemented. The YAWL designer is under development and at present only supports the specification of control-fbw aspects. The YAWL manager has not been implemented yet. Of the YAWL services only the YAWL worklist handler is realized. The implementation has been done in Java and uses XML-based standards such as XPath, XQuery, and XML Schema. The designer is implemented using Jgraph: an open source graphical library (`www.jgraph.com`). The YAWL engine relies on JDom (`www.jdom.org`) for evaluating XPath expressions, Saxon (`saxon.sourceforge.net`) for XQuery support, and Xerces (`xml.apache.org/xerces-j`) for XML schema support.

5 YAWL perspectives

This section discusses three key perspectives: (1) the control-fbw perspective, (2) the data perspective, and (3) the operational perspective. The first two perspectives are fully implemented and are supported by the YAWL engine. The operational perspective corresponds to the YAWL services identified in the previous section and is only partly realized.

5.1 Control-flow perspective

The control-fbw perspective of YAWL focuses on the ordering of tasks. The building blocks offered by YAWL have been briefly discussed in Section 3 and are depicted in

Figure 1. There are three features offered by YAWL not present in most workflow languages: (1) the OR-join task, (2) multiple instances of a task (atomic or composite), and (3) the “remove tokens” task (i.e., cancellation of a region). Therefore, we focus on these three. Let us first focus on the realization of the OR-join. Listing 1 shows

Listing 1

```
<task id="register">
    <name>Collect information
      from customer</name>
    <flowsInto>
        <nextElementRef
          id="flight"/>
        <predicate>
          /data/want_flight = ’true’
        </predicate>
        <isDefaultFlow/>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="hotel"/>
        <predicate>/data/want_hotel = ’true’</predicate>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="car"/>
        <predicate>/data/want_car = ’true’</predicate>
    </flowsInto>
    <join code="and"/>
    <split code="or"/>
    <startingMappings>
        <mapping>
            <expression query="/data/customer"/>
            <mapsTo>customer</mapsTo>
        </mapping>
    </startingMappings>
    <completedMappings>
        <mapping>
            <expression query="/data/customer"/>
            <mapsTo>customer</mapsTo>
        </mapping>
        <mapping>
            <expression query="/data/want_flight"/>
            <mapsTo>want_flight</mapsTo>
        </mapping>
        ....
    </completedMappings>
    <decomposesTo id="register"/>
</task>
```

Listing 2

```
<task id="pay">
    <name>Book flight</name>
    <flowsInto>
        <nextElementRef id="end"/>
    </flowsInto>
    <join code="or"/>
    <split code="and"/>
    <startingMappings>
        ...
    </startingMappings>
    <decomposesTo id="pay"/>
</task>
```

the definition of task *register* in Figure 2(a). The *name* element provides the name of the task, the three *flowsInto* elements correspond to the three outgoing arcs, the *join* element shows that the task is an AND-join[3], the *split* element shows that the task is an OR-split, the *startingMappings* element lists the data elements that are input to the task, the *completedMappings* element lists the data elements that are provided as output by the task, and the *decomposesTo* element refers to the actual definition of the task (which we call its "decomposition"). Note that each *task* element refers to a *decomposition* element. The *decomposition* element can define an atomic task or a composite task. Multiple *task* elements can refer to the same *decomposition* element to allow for reuse. Listing 2 shows the *task* element for the corresponding OR-join. Although from a syntactical point of view the OR-join is easy to realize, it is far from trivial to realize the corresponding functionality. In the classical XOR-join the flow continues after the first input. In the classical AND-join the flow waits for all inputs. The complicating factor is that the OR-join sometimes has to synchronize and sometimes not (or only partially). In the example shown in Figure 2(a) it is fairly easy to see when to synchronize, e.g., simply count the number of bookings enabled and then count back to zero for every booking that is completed. However, in a general sense this strategy does not work, because there can be multiple splits (of all types) corresponding to an OR-join. The semantics adopted by YAWL is that an OR-join waits until no more inputs can arrive at the join. To make sure that the semantics are well defined, i.e., have a fix-point, we exclude other OR-joins as indicated in [3]. See [14] for a more elaborate discussion on the topic. As far as we know, YAWL is the first engine implementing this strategy without adding additional constraints such as excluding loops, etc. From a performance point of view, the OR-join is quite expensive (the system needs to calculate all possible futures from the current state). To improve performance, the OR-join condition is evaluated only if strictly necessary.

A second feature which distinguishes YAWL from many existing languages is the ability to have multiple instances of atomic/composite tasks. Figure 2(b) shows an example of this. The composite task *do_itinerary_segment* has additional elements to control the number of instances (*minimum*, *maximum*, *threshold*, and *creation mode*) and elements to control the data flow. Again the syntax is fairly straightforward but the realization in the YAWL engine is not. Note that multiple instances can be nested arbitrarily deep and it becomes quite difficult to separate and synchronize instances.

A third feature worth noting is the cancellation of a region, i.e., removing tokens from selected parts of the specification. In Figure 2(c) task *cancel* contains four *removesTokens* elements to empty the part of the specification shown. The cancellation functionality is easy to realize in the engine. The biggest challenge is to allow for an easy way to indicate a region in the YAWL designer. At this point in time we are experimenting with various interaction mechanisms to allow for a generic yet intuitive way to demarcate such regions. Condition *booking_in_process* in Figure 2(c) also illustrates that YAWL supports the Deferred choice pattern. Note that the decision to cancel is made by the external entity executing task *cancel*, and not by the workflow engine. In traditional workflow systems, such "deferred" or "environment-driven" choices are not possible: all decisions are made by the system based on data as the XOR-split and OR-split constructs do. The notion of deferred choice has been adopted by new languages like BPEL (see *pick* construct in [7]) and BPML (see *choice* construct in [5]).

[3] This is not relevant in this example since there is only one incoming arc.

5.2 Data perspective

Although the initial focus of YAWL was on control fbw, it has been extended to offer full support for the data perspective. It is possible to define data elements and use them for conditional routing, for the creation of multiple instances, for exchanging information with the environment, etc. Most of the existing workfbw management systems use a propriety language for dealing with data. YAWL is one of the few languages that completely relies on XML-based standards like XPath and XQuery.

Listing 3

```
1  <rootNet id="make_trip">
2      <localVariable
3         name="customer">
4          <type>xs:string</type>
5          <initialValue>
6          Type name of customer
7          </initialValue>
8      </localVariable>
9      <localVariable name=
10       "payment_account_number">
11         <type>xs:string</type>
12     </localVariable>
13     ...
14     <localVariable name=
15       "want_flight">
16         <type>xs:boolean</type>
17     </localVariable>
18     <localVariable name=
19       "want_hotel">
20         <type>xs:boolean</type>
21     </localVariable>
22     <localVariable name=
23       "want_car">
24         <type>xs:boolean</type>
25     </localVariable>
26     <localVariable name=
27       "flightDetails">
28         <type>xs:string</type>
29     </localVariable>
30     ...
```

Listing 4

```
1  <decomposition id="register"
2     xsi:type=
3    "WebServiceBrokerFactsType">
4      <inputParam name="customer">
5          <type>xs:string</type>
6      </inputParam>
7      <outputExpression query=
8        "/data/customer"/>
9      <outputExpression query=
10       "/data/start_date"/>
11     ...
12     <outputExpression query=
13       "/data/want_flight"/>
14     ...
15     <outputParam name=
16       "customer">
17         <type>xs:string</type>
18     </outputParam>
19     <outputParam name=
20       "start_date">
21         <type>xs:dateTime</type>
22     </outputParam>
23     ...
24     <outputParam name=
25       "want_flight">
26         <type>xs:boolean</type>
27     </outputParam>
28     ...
29 </decomposition>
```

Listing 3 shows the declaration of variables for the example shown in Figure 2(a). Using the element *localVariable* it is possible to add typed variables to the top-level workfbw. For example, lines 2-7 define the variable for storing the name of the customer. The type of this variable is *string* and an initial value is defined. Each decomposition of a task into a workfbw may also have *localVariable* elements. Variables at the higher level can be passed onto the lower level. Listing 4 shows the decomposition of task *register* referred to in Figure 2(a). As shown in lines 3-6 of Listing 4, there is an input parameter named *customer*. Task *register* maps data residing at the higher level onto this input parameter at the lower level (i.e., in the decomposition) as shown in lines 23-28 of Listing 1. After completing the task, data at the lower level is passed on to the higher level. For example, lines 24-27 of Listing 4 declare the parameter *want_flight*. The data for this parameter is determined in lines 12-13 of Listing 4. After completing the decomposition, this result is mapped onto the variable *want_flight* at the higher level (see lines 35-36 of Listing 1). Note that the expression shown in line 35 of Listing 1 and the expression shown in line 12-13 of Listing 4 are XPath expressions to access a node. However, arbitrarily complex transformations are permitted here, using the full expressive power of XQuery. Moreover, parameters may be optional or mandatory.

If a task is an OR-split or XOR-split, *predicate* elements are used to specify Boolean expressions. Lines 7-10, line 15, and line 19 in Listing 1 specify the output conditions of task *register* (one for each outgoing arc). In the case of an OR-split or XOR-split, there is always a default indicated by the element *isDefaultFlow* (cf. line 10 in Listing 1). If all predicates evaluate to false, this arc is chosen, thereby acting like an "otherwise" branch. In the example of Figure 2(a), at least one of the three booking tasks should be executed. To ensure this, a flight is booked if none of the predicates evaluates to true. To allow the possibility that none of the three booking tasks is executed, one should add an arc directly from task *register* to either *pay* or the output condition. This would then be set to be the default arc. For an XOR-split each *predicate* needs to have an *ordering* attribute that is used in case multiple predicates evaluate to true. If predicates are not mutually exclusive, the one with the lowest number that evaluates to true is selected.

From the viewpoint of data, the handling of multiple instances is far from trivial. Consider for example Figure 2(b), the subprocess is executed for each segment of the itinerary and thus there is data for each segment (destination, start_date, flight_details, etc.). The number of instances created but also the maximum, minimum, and threshold may all depend on data. Data at the higher level needs to be split over the instances and after completion of the instances aggregated to data elements at the higher level. Again XQuery is used to map data from the higher level to the lower level and vice versa.

5.3 Operational perspective

As discussed in Section 4, the YAWL engine interacts with its environment by means of a collection of "YAWL services", which are responsible for handling the operational and the resource perspectives of workflow specifications, as well as for supporting communication between different YAWL engines. A deployment of the YAWL system is expected to include a number of pre-built YAWL services. The YAWL worklist handler, web service broker, and interoperability broker mentioned in Section 4 are examples of such pre-built services. Importantly, all YAWL services are required to implement a common WSDL interface[4], and reciprocally, the YAWL engine provides a single interface for all YAWL services. These interfaces define message types for:

– Atomic task decomposition management: registering and un-registering atomic task decompositions into YAWL services.

[4] We use the terminology of WSDL version 2.0 rather than version 1.1.

– Atomic task instance management: creating task instances, notifying the start and completion (whether successful or not) of task instances, canceling task instances, and probing the status of a task instance.
– Workfbw instance management: creating, monitoring, and interacting with workfbw instances.
– YAWL services connection management: registering and un-registering YAWL services, reporting and probing the availability of YAWL services.

When a new YAWL workfbw specification is deployed, the YAWL engine registers each of the atomic task decompositions included in this specification, with at least one YAWL service. Each task decomposition indicates the YAWL service(s) with which it has to be registered. In the setting of the travel preparation example, one possible scenario is that the tasks *register*, *flight*, and *hotel* are to be registered with the worklist service, while the task *pay* is to be registered with the YAWL web services broker (e.g. the payment is handled by an external payment service). If these registrations are successful, the YAWL engine is then able to create instances of these tasks. Unsuccessful task registrations lead to errors and the YAWL engine reports back these errors to the YAWL designer or the YAWL manager. The deployment of a workfbw specification is only considered to be successful if all the registrations of task decompositions in the specification, are successful. Otherwise, the deployment is aborted and any registered task decompositions are unregistered.

As a minimum, a task decomposition specifies the task's input and output data types. It may specify other information depending on the nature of the YAWL service with which the task will be registered. *In the case of tasks that are registered with a worklist service*, the task decomposition must specify the role(s) that are able to view instances of this task and the role(s) that are able to pick instances of this task from the worklist.

In the case of a web service broker, the task decomposition must include:

1. A WSDL interface and SOAP binding of the web service *WS* that the broker must invoke when dispatching an instance of this task.
2. A mapping between the task management operations of the YAWL services interface and the operations in the interface of *WS*. Using this information, the web service broker can exploit the functionality provided by the Web Services Invocation Framework (http://ws.apache.org/wsif) in order to interact with *WS*.

In the setting of the travel preparation workfbw, and assuming that the task *pay* is delegated to a payment service (say *PS*), the decomposition of this task must provide the WSDL interface and binding for *PS*, and a table indicating that for example:

– Operation *CreateTaskInstance* of the common YAWL services interface is mapped to operation *InitiateOnlinePayment* of *PS* service.
– Operation *OnlinePaymentInitiated* of *PS* maps to operation *TaskStarted* of the YAWL services interface.
– Operation *OnlinePaymentCompleted* of *PS* maps to operation *TaskCompleted* of the YAWL services interface.
– Operation *CancelTask* of the common YAWL services interface maps to operation *CancelPayment* of *PS*.

These mappings should also specify how the input data of one operation maps to the input data of the other operation, and same for the output data.

To be registered with a YAWL interoperability broker service, a task decomposition *TD* must specify: (i) the identifier of the YAWL engine to which instances of this task will be delegated; (ii) the name of the YAWL process to be instantiated when an instance of the task is created; (iii) a mapping between the input data of the *CreateTaskInstance* operation and the input data of the process to be instantiated; and (iv) a similar mapping for the output data. When an instance of *TD* is created, the interoperability broker creates an instance of the designated process in a possibly remote YAWL engine (using the workfbw instance management operations of the YAWL services interfaces). When this workfbw instance completes, the interoperability broker collects the output data, converts them using the mapping given when the task decomposition was registered, and returns them to the YAWL engine that triggered the instantiation of *TD*. Note that this process interoperability model assumes that no communication occurs between the creation and completion of a process instance. It is envisaged that the YAWL system will be extended to support communication between running process instances.

6 Example and on-line demonstration

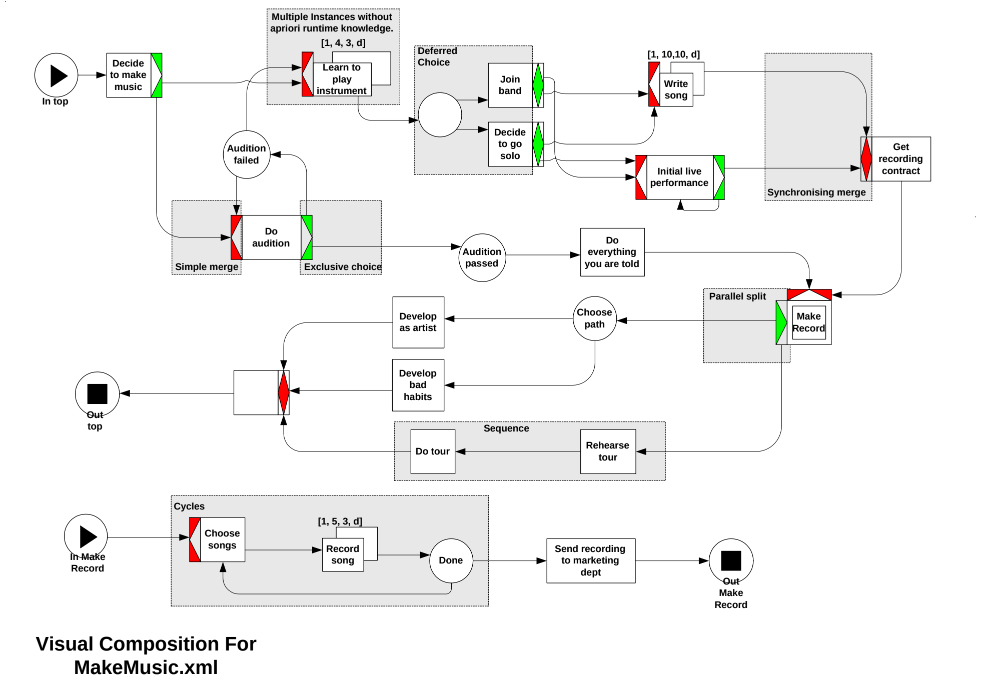
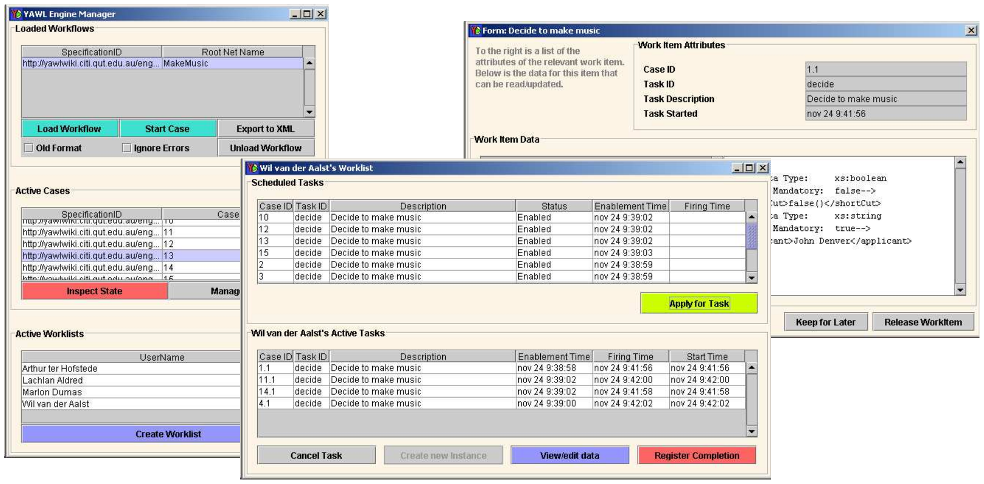


Fig. 4. Example of a YAWL process.

This section illustrates the current implementation of YAWL using a small example that can be downloaded and run from the YAWL site `www.citi.qut.edu.au/yawl/`. Figure 4 shows the life-cycle of a musician from the viewpoint of a record company. The goal of this playful example is not to show a realistic business scenario but an easy to understand example showing the main concepts. The top-level process starts with a choice between doing an audition or first learning to play an instrument (Pattern 4: Exclusive choice). The musician can learn multiple instruments in parallel (Pattern 15: Multiple instances without a priori runtime knowledge) followed by the decision to join a band or to go solo (Pattern 16: Deferred choice). In both cases, multiple songs may be written (again Pattern 15) and/or a live performance is given after which the musician gets a contract (Pattern 6: Multi-choice/Pattern 7: Synchronizing merge). The audition can fail and, if so, the musician tries again or continues learning to play instruments (Pattern 5: Simple merge/Pattern 16: Deferred choice/Pattern 10: Arbitrary cycles). Eventually the musician ends up making a record. This is modeled by a composite task (*Make Record*) containing a loop in which multiple songs can be recorded (see lower-level process in Figure 4). The subprocess uses Pattern 5: Simple merge, Pattern 16: Deferred choice, and Pattern 15: Multiple instances without a priori runtime knowledge. After completing the subprocess in parallel a choice is made and a sequence is executed, followed by a synchronization (Pattern 2: Parallel split/Pattern 16: Deferred choice/Pattern 1: Sequence/Pattern 7: Synchronizing merge) thus completing the YAWL specification.

Figure 4 does not show the data perspective (which is specified separately). The data perspective of this workflow specification states that musicians have a name, songs have a title, etc. Additionally, in the case of the composite task *Make Record* which will be instantiated multiple times, the data perspective specifies the effective parameters that will be passed to each instance (e.g. an expression for computing the actual name and other properties of a given song).



SpecificationID	Root Net Name
http://yawlwiki.citi.qut.edu.au/eng...	MakeMusic

SpecificationID	Case
http://yawlwiki.citi.qut.edu.au/eng...	11
http://yawlwiki.citi.qut.edu.au/eng...	12
http://yawlwiki.citi.qut.edu.au/eng...	13
http://yawlwiki.citi.qut.edu.au/eng...	14

UserName
Arthur ter Hofstede
Lachlan Aldred
Marlon Dumas
Wil van der Aalst

Case ID	Task ID	Description	Status	Enablement Time	Firing Time
10	decide	Decide to make music	Enabled	nov 24 9:39:02	
12	decide	Decide to make music	Enabled	nov 24 9:39:02	
13	decide	Decide to make music	Enabled	nov 24 9:39:02	
15	decide	Decide to make music	Enabled	nov 24 9:39:03	
2	decide	Decide to make music	Enabled	nov 24 9:38:59	
3	decide	Decide to make music	Enabled	nov 24 9:38:59	

Case ID	Task ID	Description	Enablement Time	Firing Time	Start Time
1.1	decide	Decide to make music	nov 24 9:38:58	nov 24 9:41:56	nov 24 9:41:56
11.1	decide	Decide to make music	nov 24 9:39:02	nov 24 9:42:00	nov 24 9:42:00
14.1	decide	Decide to make music	nov 24 9:39:02	nov 24 9:41:58	nov 24 9:41:58
4.1	decide	Decide to make music	nov 24 9:39:00	nov 24 9:42:02	nov 24 9:42:02

To the right is a list of the attributes of the relevant work item. Below is the data for this item that can be read/updated.

Case ID	1.1
Task ID	decide
Task Description	Decide to make music
Task Started	nov 24 9:41:56

```
a Type:     xs:boolean
 Mandatory:  false-->
Cut>false()</shortCut>
a Type:     xs:string
 Mandatory:  true-->
cant>John Denver</applicant>
```

Fig. 5. Screenshots of the current YAWL manager, worklist handler, and a data entry form.

Figure 5 shows some screenshots of the tool while executing several instances of the YAWL specification of Figure 4. The worklist of 'Wil van der Aalst' is shown in the bottom window. The left window shows the current YAWL manager showing the active workflows, active cases, and users. The right window shows an interface to enter data. This is still rather primitive. Future versions of YAWL are expected to support interactive forms.

7 Conclusion

In this paper we presented the design and implementation of the YAWL system. The YAWL system fully supports the YAWL language which is based on an analysis of more than 30 workflow systems, languages and standards. The expressiveness of YAWL leads to challenging implementation problems such as dealing with multiple instances, advanced synchronization mechanisms, and cancellation capabilities. We consider the current version of YAWL as a proof of concept for the language introduced in [3]. In our opinion any proposed language should be supported by at least a running prototype in addition to a formal definition [3]. Too many standards and languages have been proposed which turn out to have semantic problems.

At this point in time we are implementing the architecture shown in Figure 3. As indicated in Section 4, the current version of the implementation provides the basic functionality of a workflow system, but not all the components of the architecture have been realized. One of the most challenging issues is to fine-tune the definition of the interactions between the engine and the YAWL services. Other directions for future effort include testing the language and system against complex application scenarios and studying the possibility of using YAWL as an intermediate language to facilitate the development of mappings between various business process execution languages.

Both the executable and sources of YAWL can be downloaded from `www.citi.qut.edu.au/yawl`. YAWL is an open source initiative and therefore welcomes contributions from third parties which could range from dedicated YAWL services to alternative graphical editors or even alternative implementations of the engine itself.

Acknowledgments. The authors wish to thank Lindsay Bradford, David Edmond, Nick Russell, Eric Verbeek, and Moe Wynn for their contributions to the YAWL effort. This work is partly funded by an ARC Discovery Grant "Expressiveness Comparison and Interchange Facilitation between Business Process Execution Languages".

References

1. 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.
2. W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In K. Jensen, editor, *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, volume 560 of *DAIMI*, pages 1–20, Aarhus, Denmark, August 2002. University of Aarhus.
3. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. Accepted for publication in *Information Systems*, and also available as QUT Technical report, FIT-TR-2003-04, Queensland University of Technology, Brisbane, 2003.

4. 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.
5. A. Arkin et al. Business Process Modeling Language (BPML), Version 1.0, 2002.
6. Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
7. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.0. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2002.
8. 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 Unifi ed Modeling Language (UML01)*, volume 2185 of *LNCS*, pages 76–90, Toronto, Canada, October 2001. Springer Verlag.
9. L. Fischer, editor. *Workflow Handbook 2001, Workflow Management Coalition*. Future Strategies, Lighthouse Point, Florida, 2001.
10. IBM. *IBM MQSeries Workflow - Getting Started With Buildtime*. IBM Deutschland Entwicklung GmbH, Boeblingen, Germany, 1999.
11. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
12. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.
13. 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.
14. E. Kindler. On the Semantics of EPCs: A Framework for Resolving the Vicious Circle (Extended Abstract). In M. Nüttgens and F.J. Rump, editors, *Proceedings of the GI-Workshop EPK 2003: Business Process Management using EPCs*, pages 7–18, Bamberg, Germany, October 2003. Gesellschaft für Informatik, Bonn.
15. P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997.
16. Open Source Workflow Engines Written in Java (maintained by Carlos E. Perez). http://www.manageability.org/blog/stuff/workflow_in_java.
17. C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.
18. M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
19. 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.
20. 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.
21. 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.
22. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In I.Y. Song, S.W. Liddle, T.W. Ling, and P. Scheuermann, editors, *22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer-Verlag, Berlin, 2003.