Workflow Patterns

W.M.P. van der Aalst[1]*, A.H.M. ter Hofstede[2]†, B. Kiepuszewski[3]†, and A.P. Barros[4]‡

[1]*Department of Technology Management, Eindhoven University of Technology*
GPO Box 513, NL-5600 MB Eindhoven, The Netherlands, e-mail: w.m.p.v.d.aalst@tm.tue.nl;
[2]*Cooperative Information Systems Research Centre, Queensland University of Technology*
GPO Box 2434, Brisbane Qld 4001, Australia, e-mail: arthur@icis.qut.edu.au;
[3]*Mincom Pty Ltd, GPO Box 1397, Brisbane Qld 4001, Australia, e-mail: bartek@mincom.com;*
[4]*Distributed Systems Technology Centre, The University of Queensland*
Brisbane Qld 4072, Australia, e-mail: abarros@dstc.edu.au.

Abstract

Differences in features supported by the various contemporary commercial workflow management systems point to different insights of *suitability* and different levels of *expressive power*. The challenge, which we undertake in this paper, is to systematically address workflow requirements, from basic to complex. Many of the more complex requirements identified, recur quite frequently in the analysis phases of workflow projects, and present grave uncertainties when looking at current products. Requirements for workflow languages are indicated through workflow *patterns*. In this context, patterns address business requirements in an imperative workflow style expression, but are removed from specific workflow languages. The paper describes a number of workflow patterns addressing what we believe identify comprehensive workflow functionality. These patterns provide the basis for an in-depth comparison of a number of commercially available workflow management systems. As such, this paper can be seen as the academic response to evaluations made by prestigious consulting companies. Typically, these evaluations hardly consider the workflow modeling language and routing capabilities and focus more on the purely technical and commercial aspects.

*Part of this work was done at CTRG (University of Colorado, USA) during a sabbatical leave.

†This research was partially supported by an ARC SPIRT grant "Component System Architecture for an Open Distributed Enterprise Management System with Configurable Workflow Support" between QUT and Mincom.

‡Part of this work was supported by CITEC, an agency within the Queensland State Government.

1 Introduction

Background

Workflow technology continues to be subjected to on-going development in its traditional application areas of business process modeling and business process coordination, and now in emergent areas of component frameworks and inter-workflow, business-to-business interaction. Addressing this broad and rather ambitious reach, a large number of workflow products, mainly workflow management systems (WFMS), are commercially available, which see a large variety of languages and concepts based on different paradigms (see e.g. [Aal98a, AH00, EN93, GHS95, JB96, Kou95, LR99, Law97, Sch96, WFM96, DKTS98]).

As current provisions are compared and as newer concepts and languages are embarked upon, it is striking how little, other than standards glossaries, is available for central reference. One of the reasons attributed to the lack of consensus of what constitutes a workflow specification is the organizational level of definition imparted by workflows. The absence of a universal organizational "theory", it is contended, explains and ultimately justifies the major differences - opening up a "horses for courses" diversity for different business domains. What is more, the comparison of different workflow products winds up being more of a dissemination of products and less of a critique of workflow language capabilities - "bigger picture" differences of workflow specifications are highlighted, as are technology, typically platform dependent, issues.

Workflow specifications can be understood, in a broad sense, from a number of different perspectives (see [JB96]). The *control-flow* perspective (or process) perspective describes activities and their execution ordering through different constructors, which permit flow of execution control, e.g. sequence, splits, parallelism and join synchronization. Activities in elementary form are atomic units of work, and in compound form modularize an execution order of a set of activities. The *data perspective* layers business and processing data on the control perspective. Business documents and other objects which flow between activities, and local variables of the workflow, qualify in effect pre- and post-conditions of activity execution. The *resource perspective* provides an organizational structure anchor to the workflow in the form of human and device roles responsible for executing activities. The *operational* perspective describes the elementary actions executed by activities, where the actions map into underlying applications. Typically, (references to) business and workflow data are passed into and out of applications through activity-to-application interfaces, allowing manipulation of the data within applications.

Clearly, the control flow perspective provides an essential insight into a workflow specification's effectiveness. The data flow perspective rests on it, while the organizational and operational perspectives are ancillary. If workflow specifications are to be extended to meet newer processing requirements, control flow constructors require a fundamental insight and analysis. Currently, most workflow languages support the basic constructs of sequence, iteration, splits (AND and OR) and joins (AND and OR) - see [Law97]. However, the interpretation of even

these basic constructs is not uniform and it is often unclear how more complex requirements could be supported. Indeed, vendors are afforded the opportunity to recommend implementation level "hacks" such as database triggers and application event handling. The result is that neither current capabilities nor an insight into newer requirements is advanced.

Problem

Even without formal qualification, the distinctive features of different workflow languages allude to fundamentally different semantics. Some languages allow multiple instances of the same activity type at the same time in the same workflow context while others do not. Some languages structure loops with one entry point and one exit point, while in others loops are allowed to have arbitrary entry and exit points. Some languages require explicit termination activities for workflows and their compound activities while in others termination is implicit. Such differences point to different insights of *suitability* and different levels of *expressive power*.

The challenge, which we undertake in this paper, is to systematically address workflow requirements, from basic to complex, in order to 1) identify useful routing constructs and 2) to establish to what extent these requirements are addressed in the current state of the art. Many of the basic requirements identify slight, but subtle differences across workflow languages, while many of the more complex requirements identified in this paper, in our experiences, recur quite frequently in the analysis phases of workflow projects, and present grave uncertainties when looking at current products. Given the fundamental differences indicated above, it is tempting to build extensions to one language, and therefore one semantic context. Such a strategy is rigorous and its results would provide a detailed and unambiguous view into what the extensions entail. Our strategy is more practical. We wish to draw a more *broader* insight into the implementation consequences for the big and potentially big players. With the increasing maturity of workflow technology, workflow language extensions, we feel, should be levered across the board, rather than slip into "yet another technique" proposals.

Approach

We indicate requirements for workflow languages through workflow *patterns*. As described in [RZ96], a pattern "is the abstraction from concrete form which keeps recurring in specific non-arbitrary contexts". Gamma et al. [GHJV95] 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. [Fow97]).

For our purpose, patterns address business requirements in an imperative workflow style expression, but are removed from specific workflow languages. Thus they do not claim to be the only way of addressing the business requirements. Nor are they "alienated" from the workflow

approach, thus allowing a potential mapping to be positioned closely to different languages and implementation solutions. Along the lines of [GHJV95], patterns are described through: conditions that should hold for the pattern to be applicable; examples of business situations; problems, typically semantic problems, of realization in current languages; and implementation solutions.

Our claim is that the workflow patterns identified in this paper are comprehensive with respect to currently available workflow languages. Some of the patterns can be mapped into constructs of existing languages fairly straightforwardly, others can be realized through the implementation level, while there also exist patterns that are supported only by a small minority of the workflow management systems. No contemporary workflow management system supports all patterns.

The organization of this paper is as follows. First, we describe the workflow patterns, then we present the comparison of contemporary workflow management systems using the patterns (except the most elementary ones, as they are supported by all workflow management systems). Finally, we conclude the paper and identify issues for further research.

2 Workflow Patterns

The design patterns range from fairly simple constructs present in any workflow language to complex routing primitives not supported by today's generation of workflow management systems. We will start with the more simple patterns. Since these patterns are available in the current workflow products we will just give a (a) *description*, (b) *synonyms*, and (c) some *examples*. In fact, for these rather basic constructs, the term "workflow pattern" is not very appropriate. However, for the more advanced routing constructs we also identify (d) the *problem* and (e) potential *solutions*. The problem component of a pattern describes why the construct is hard to realize in many of the workflow management systems available today. The solution component describes how, assuming a set of basic routing primitives, the required behavior can be realized. For these more complex routing constructs the term "pattern" is more justified since non-trivial solutions are given for practical problems encountered when using today's workflow technology.

Before we present the patterns, we first introduce some of the terms that will be used throughout this paper. The primary task of a workflow management system is to enact case-driven business processes by allowing workflow models to be specified, executed, and monitored. *Workflow process definitions* (workflow schemas) are defined to specify which *activities* need to be executed and in what order (i.e. the *routing* or *control flow*). An elementary activity is an atomic piece of work. Workflow process definitions are instantiated for specific *cases* (i.e. workflow instances). Examples of cases are: a request for a mortgage loan, an insurance claim, a tax declaration, an order, or a request for information. Since a case is an instantiation of a process definition, it corresponds to the execution of concrete work according to the specified routing. Activities are connected through *transitions* and we use the notion of a *thread*

of execution control for concurrent executions in a workflow context. Activities are undertaken by *roles* which define organizational entities, such as humans and devices. *Control data* are data introduced solely for workflow management purposes, e.g. variables introduced for routing purposes. *Production data* are information objects (e.g. documents, forms, and tables) whose existence does not depend on workflow management. Elementary actions are performed by roles while executing an activity for a specific case, and are executed using *applications* (ranging from a text editor to custom built applications to perform complex calculations).

2.1 Basic Control Flow Patterns

In this section patterns capturing elementary aspects of process control are discussed. The first pattern we consider is the sequence.

Pattern 1 (Sequence)
Description An activity in a workflow process is enabled after the completion of another activity in the same process.
Synonyms Sequential routing, serial routing.
Examples

- Activity *send_bill* is executed after the execution of activity *send_goods*.
- An insurance claim is evaluated after the client's file is retrieved.
- Activity *add_air_miles* is executed after the execution of activity *book_flight*.

Problem Does not cause any specific problems.
Solutions

- The sequence pattern is used to model consecutive steps in a workflow process and is directly supported by each of the workflow management systems available.

□

The next two patterns can be used to accommodate for parallel routing.

Pattern 2 (Parallel Split)
Description A point in the workflow process where a single thread of control splits into multiple treads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order.
Synonyms AND-split, parallel routing, fork.
Examples

- The execution of the activity *payment* enables the execution of the activities *ship_goods* and *inform_customer*.

- After registering an insurance claim two parallel subprocesses are triggered: one for checking the policy of the customer and one for assessing the actual damage.

Problem Does not cause any specific problems.
Solutions

- All workflow engines available support constructs for the implementation of parallel execution patterns. One can identify two basic approaches: explicit AND-splits and implicit AND-splits. Workflow engines supporting the explicit AND-split construct (e.g. Visual WorkFlo) define a routing node with more than one outgoing transition which will be enabled as soon as the routing node gets enabled. Workflow engines supporting implicit AND-splits (e.g. MQSeries/Workflow) do not provide special routing constructs - each activity can have more than one outgoing transition and each transition has associated conditions. To achieve parallel execution the workflow designer has to make sure that multiple conditions associated with outgoing transitions of the node evaluate to True (this is typically achieved by leaving the conditions blank).

□

Pattern 3 (Synchronization)
Description A point in the workflow process where multiple parallel subprocesses/activities converge into one single thread of control, thus synchronizing multiple threads.
Synonyms AND-join, rendezvous, synchronizer.
Examples

- Activity *archive* is enabled after the completion of both activity *send_tickets* and activity *receive_payment*.

- Insurance claims are evaluated after the policy has been checked and the actual damage has been assessed.

Problem Does not cause any specific problems.
Solutions

- All workflow engines available support constructs for the implementation of this pattern. Typically there is a special synchronizing construct available. In some rare cases, synchronization has to be implemented by providing a special start condition for an activity that has more than one incoming transition.

- When an explicit synchronization construct is available (synchronizer), it will typically have more than one incoming transition and exactly one outgoing transition. We have experienced that even though the notion of synchronization of two concurrently running threads is intuitively simple, the actual semantics of the synchronizer differs from product to product. Leading workflow products such as Staffware, Verve and HP ChangeEngine for example, support a synchronizer notion whereby multiple triggering by the same activity is ignored. If for example an activity C is preceded by a synchronizer having

transitions from activities A and B as input, this synchronizer will ignore termination of instances of activity A if it has already seen one such instance and is waiting for the termination of an instance of activity B. Another approach would be to simply keep track of the number of "extra" instances of activity A that terminated while waiting for activity B and try to match them later with corresponding instances of activity B.

□

The next two patterns are used to specify conditional routing. In contrast to parallel routing only one selected thread of control is activated.

Pattern 4 (Exclusive Choice)
Description A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen.
Synonyms XOR-split, conditional routing, switch, decision.
Examples

- Activity *evaluate_claim* is followed by either *pay_damage* or *contact_customer*.

- Based on the workload, a processed tax declaration is either checked using a simple administrative procedure or a thorough evaluation by a senior employee.

Problem Does not cause any specific problems.
Solutions

- Similarly to Pattern 2 there are two basic strategies - some workflow engines provide an explicit construct for the implementation of the exclusive choice pattern (e.g. Staffware, Visual WorkFlo), while in others (MQSeries/Workflow, Verve) the workflow designer has to emulate the exclusiveness of choice by a selection of transition conditions.

□

Pattern 5 (Simple Merge)
Description A point in the workflow process where two or more alternative branches come together without synchronization. In other words the merge will be triggered once any of the incoming transitions are triggered.
Synonyms XOR-join, asynchronous join, merge.
Examples

- Activity *archive_claim* is enabled after either *pay_damage* or *contact_customer* is executed.

- After the payment is received or the credit is granted the car is delivered to the customer.

Problem This pattern causes no particular problems in case at most one of the incoming branches can be active at any point in time, which can be statically enforced (if this is not the

case we refer to Pattern 8).
Solutions

- Given that we are assuming that only one of the incoming transitions can be triggered at one time, this is a straightforward situation and all workflow engines support a construct that can be used to implement the simple merge. It is interesting to note here that some of the languages impose a certain level of structuredness to automatically guarantee that not more than one incoming transition can be triggered. Visual WorkFlo for example, requires the merge construct to always be preceded by an exclusive choice construct. In other languages the workflow designers are responsible themselves for the design not having multiple incoming transitions that can be triggered.

□

2.2 Advanced Branching and Synchronization Patterns

In this section the focus will be on more advanced patterns for branching and synchronization. As opposed to the patterns in the previous section, these patterns do not have straightforward support in most workflow engines. Nevertheless, they are quite common in real-life business scenarios.

Pattern 4 assumes that exactly one of the alternatives is selected and executed, i.e. it corresponds to an *exclusive* OR. Sometimes it is useful to deploy a construct which can choose multiple alternatives from a given set of alternatives. Therefore, we introduce the (inclusive) multi-choice.

Pattern 6 (Multi-choice)
Description A point in the workflow process where, based on a decision or workflow control data, one or more branches are chosen.
Synonyms Conditional routing, selection, OR-split.
Examples

- After executing the activity *evaluate_damage* the activity *contact_fire_department* or the activity *contact_insurance_company* is executed. At least one of these activities is executed. However, it is also possible that both need to be executed.

Problem In many workflow management systems one can specify conditions on the transitions. In these systems, the OR-split can be captured directly. However, there are several workflow management systems which do not offer the possibility to specify conditions on transitions and only offer pure AND-split and XOR-split building blocks (e.g. Staffware).
Solutions

- As stated, for workflow languages that assign transition conditions to each transition (e.g. Verve, MQSeries/Workflow, Forté Conductor) the implementation of the multi-choice is straightforward. The workflow designer simply specifies desired conditions for

each transition. It may be noted that the multi-choice pattern generalizes the parallel split (Pattern 2) and the exclusive choice (Pattern 4).

- For languages that supply only constructs to implement the parallel split and the exclusive choice, the implementation of the multi-choice has to be achieved through using a combination of the two. Each possible branch is preceded by an XOR-split which decides, based on control data, either to activate the branch or to bypass it. All XOR-splits are activated by one AND-split. This AND-split can also be used to set the control data that is used in the XOR-splits.

- A solution similar to the previous one is obtained by reversing the order of patterns 2 and 4. For each set of branches which can be activated in parallel, one AND-split is added. All AND-splits are preceded by one XOR-split which activates the appropriate AND-split. Note that, typically, not all combinations of branches are possible. Therefore, this solution may lead to a more compact workflow specification. Both solutions are depicted in Figure 1.

□

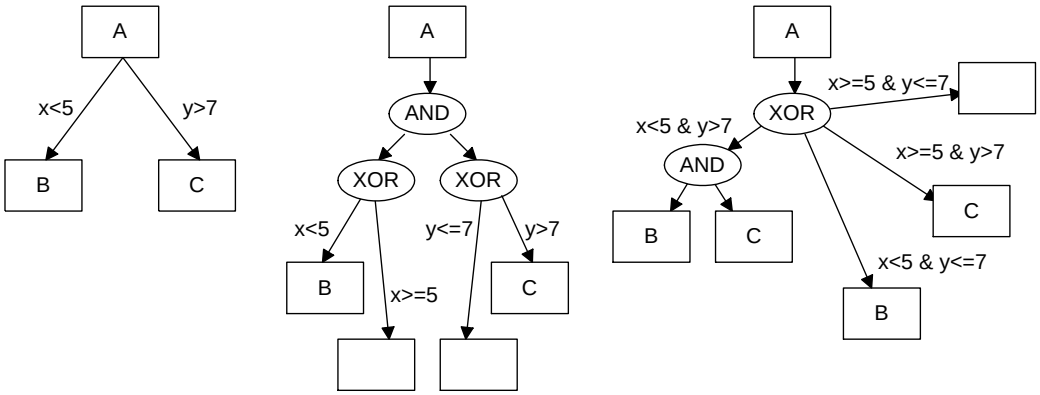

Figure 1: Design patterns for the multi-choice

The OR-split can be handled quite easily by today's workflow products. Unfortunately, the implementation of the corresponding merge construct (OR-join) is much more difficult to realize. The OR-join has the capability to synchronize parallel flows and to merge alternative flows. The difficulty is to decide when to synchronize and when to merge. Synchronizing alternative flows leads to potential deadlocks and merging parallel flows may lead to the undesirable multiple execution of activities.

Pattern 7 (Synchronizing merge)
Description A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place.

If only one path is taken, the alternative branches should reconverge without synchronization.
Synonyms Synchronizing join
Examples

- Extending the example of Pattern 6, after either or both of the activities *contact_fire_department* and *contact_insurance_company* have been completed (depending on whether they were executed at all), the activity *submit report* needs to be performed (exactly once).

Problem The main difficulty with this pattern is to decide when to synchronize and when to merge. Synchronizing alternative flows leads to potential deadlocks and merging parallel flows may lead to unwanted, multiple execution of the activity that follows the standard OR-join construct.
Solutions

- The two workflow engines known to the authors that provide a straightforward construct for the realization of this pattern are MQSeries/Workflow and InConcert. As noted earlier, if a synchronizing merge follows an OR-split and more than one outgoing transition of that OR-split can be triggered, it is not until runtime that we can tell whether or not synchronization should take place. MQSeries/Workflow works around that problem by passing a False token for each transition that evaluates to False and a True token for each transition that evaluates to True. The merge will wait until it receives tokens from each incoming transition. InConcert does not use a False token concept. Instead it passes a token through every transition in a graph. This token may or may not enable the execution of an activity depending on the entry condition. This way every activity having more than one incoming transition can expect that it will receive a token from each one of them, thus deadlock cannot occur. The careful reader may note that these evaluation strategies require that the workflow process does not contain cycles.

- In all other workflow engines the implementation of the synchronizing merge is not straightforward. The common design pattern is to avoid the explicit use of the OR-split that may trigger more than one outgoing transition and implement it as a combination of AND-splits and XOR-splits (see Pattern 6). This way we can easily synchronize corresponding branches by using AND-join and XOR-join constructs.

□

The next two patterns aim to address the problem mentioned in Pattern 5, that is the situation when more than one incoming transition of a merge is being activated.

Pattern 8 (Multi-merge)
Description Multi-merge is a point in a workflow process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started *once for every incoming branch that gets activated.*
Examples

- Sometimes two or more parallel branches share the same ending. Instead of replicating this (potentially complicated) process for every branch, a multi-merge can be used. A simple example of this would be two activities *audit_application* and *process_application* running in parallel which should both be followed by an activity *close_case*.

Problem Most workflow engines (e.g. Staffware, HP Changengine, I-Flow) will not generate the second instance of an activity if the first instance is still running. Notable exceptions are Verve Worflow and Forté Conductor.

Solutions

- If the multi-merge is not part of a loop, the common design pattern for languages that are not able to create more than one active instance of an activity is to replicate this activity in the workflow model (see Figure 2 for a simple example). If the multi-merge is part of the loop, then typically the number of instances of an activity following the multi-merge is not known during design time. For a typical solution to this problem, see patterns 14 and 15.

□

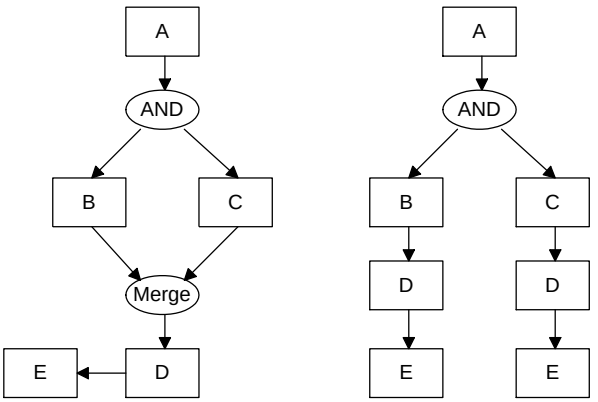

Figure 2: Typical implementation of multi-merge pattern

The next pattern can be seen as the converse of the multi-merge.

<u>Pattern</u> 9 (Discriminator)

Description The discriminator is a point in a workflow process that waits for a number of incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and "ignores" them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again.

Examples

- A paper needs to be sent to external reviewers. The paper is accepted if both reviews are positive. But if the first review that arrives is negative, the author(s) should be notified without having to wait for the second review.

- To improve query response time, a complex search is sent to two different databases over the Internet. The first one that comes up with the result should proceed the flow. The second result is ignored.

Problem As mentioned in Pattern 8, some workflow engines (e.g. Staffware, HP ChangeEngine, I-Flow) will not generate the second instance of an activity if the first instance is still active. However, this does *not* provide a solution for the discriminator since once the first instance of the activity finishes, the second instance *will* be created.

Solutions

- There is a special construct that implements the discriminator semantics in Verve.

- The discriminator semantics can be implemented in products supporting *Custom Triggers* (see Pattern 10 for details).

- In all other workflow engines the discriminator semantics is hard or impossible to implement. The common design pattern is to use Cancel Activity (see Pattern 21). Once the first instance of the activity following the discriminator is created, the activities of the incoming branches that still have not completed can be canceled. This way the second instance of the activity following the discriminator will not be created. This pattern is shown in the example of Figure 3. The problem with this solution is that if activities B and C are performed concurrently, activity D may still end up being executed twice. Moreover, the original semantics of the discriminator is to allow both B and C to finish. In this solution either B or C will get canceled.

- The Petri-net semantics of the discriminator as presented in [HK99] shows that the discriminator inherently is a non-free choice construct (see e.g. [DE95]) and hence cannot be captured exactly by workflow products supporting free choice behavior exclusively.

□

The following pattern can be seen as a generalization of the basic discriminator.

Pattern 10 (N-out-of-M Join)

Description N-out-of-M Join is a point in a workflow process where M parallel paths converge into one. The subsequent activity should be activated once N paths have completed. Completion of all remaining paths should be ignored. Similarly to the discriminator, once all incoming branches have "fired", the join resets itself so that it can fire again

Synonyms Partial join (cf. [CCPP95]), discriminator, custom join.

Examples

- A paper needs to be sent to three external reviewers. Upon receiving two reviews the paper can be processed. The third review can be ignored [CCPP95].

Problem Most of the workflow products do not provide constructs that would allow for straightforward implementation of the N-out-of-M Join.

Solutions

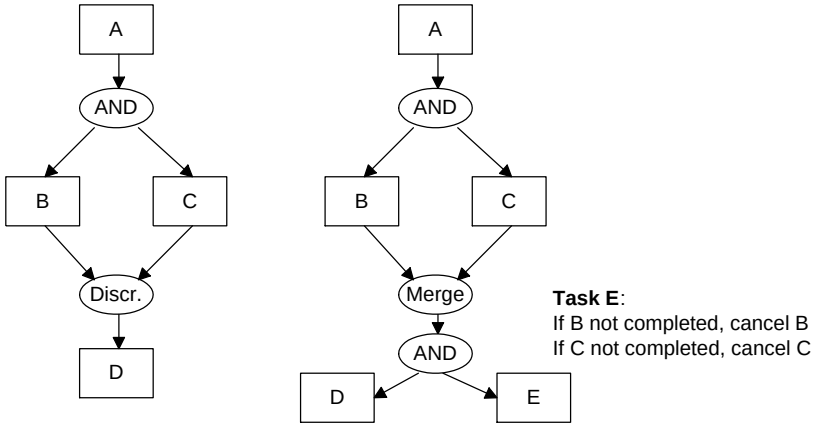

Figure 3: Design pattern for discriminator

- Some workflow engines (e.g. Forté Conductor) provide support for *Custom Triggers*. A custom trigger can be defined for an activity that has more than one incoming transition. It defines the condition, typically using some internal script language, that would activate the activity when evaluated to True. Such a script can be easily used to define the semantics equivalent to that of an N-out-of-M Join. The downside of this approach is that the semantics of the join using custom triggers is impossible to determine without carefully examining underlying trigger scripts which result in less suitable and hard to understand models.

- By combining patterns 9 and 3 one can achieve the desired semantics although the workflow definition becomes large and complex. An example of a 2-out-of-3 join is shown in Figure 4.

□

2.3 Structural Patterns

Different workflow management systems impose different restrictions on their workflow models. These restrictions (e.g. arbitrary loops are not allowed, only one final node should be present etc) are not always natural from a modeling point of view and tend to restrict the specification freedom of the business analyst. As a result, business analysts either have to conform to the restrictions of the workflow language from the start, or they model their problems freely and transform the resulting specifications afterwards. A real issue here is that of suitability. In many cases the resulting workflows may be unnecessarily complex which impacts end-users who may wish to monitor the progress of their workflows. In this section two patterns are presented which illustrate typical restrictions imposed on workflow specifications and their consequences.

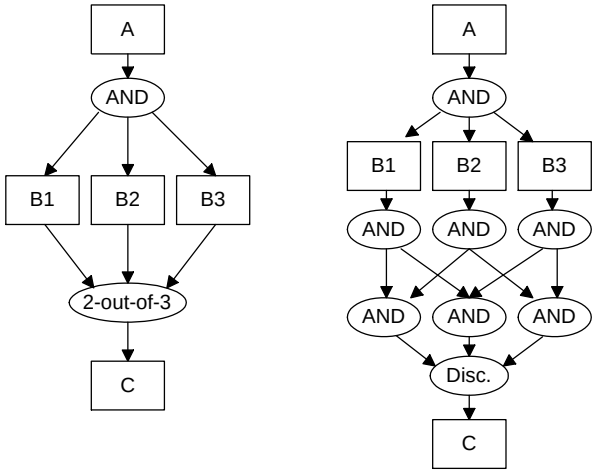

Figure 4: Implementation of 2-out-of-3-join using simple discriminator

Virtually every workflow engine has constructs that support the modeling of loops. Some of the workflow engines provide support only for what we will refer to as *structured cycles*. Structured cycles can have only one entry point to the loop and one exit point from the loop and they cannot be interleaved. They can be compared to WHILE loops in programming languages while arbitrary cycles are more like GOTO statements. This analogy should not deceive the reader though into thinking that arbitrary cycles are not desirable as there are two important differences here with "classical" programming languages: 1) the presence of parallelism which in some cases makes it impossible to remove certain forms of arbitrariness and 2) the fact that the removal of arbitrary cycles may lead to workflows that are much harder to interpret (and as opposed to programs, workflow specifications also have to be understood at runtime by their users).

Pattern 11 (Arbitrary Cycles)
Description A point in a workflow process when one or more activities can be done repeatedly.
Synonyms Loop, iteration, cycle.
Examples

- Most of the initial workflow models at the analysis stage contain arbitrary cycles (if they contain cycles at all).

Problem Some of the workflow engines do not allow arbitrary cycles - they have support for structured cycles only, either through the decomposition construct (MQSeries/Workflow, InConcert) or through a special loop construct (Visual WorkFlo, SAP R/3).
Solutions

- Arbitrary cycles can typically be converted into structured cycles unless they contain

one of the more advanced patterns such as multiple instances (see Pattern 14). The conversion is done either through auxiliary variables or through node repetition. A detailed analysis of such transformations and the extent to which they are possible, can be found in [KHB00]. Figure 5 provides an example of an arbitrary workflow converted to a structured workflow. Note that auxiliary variables Φ and Θ are required as we may not know which activities in the original workflow set the values of β and χ.

□

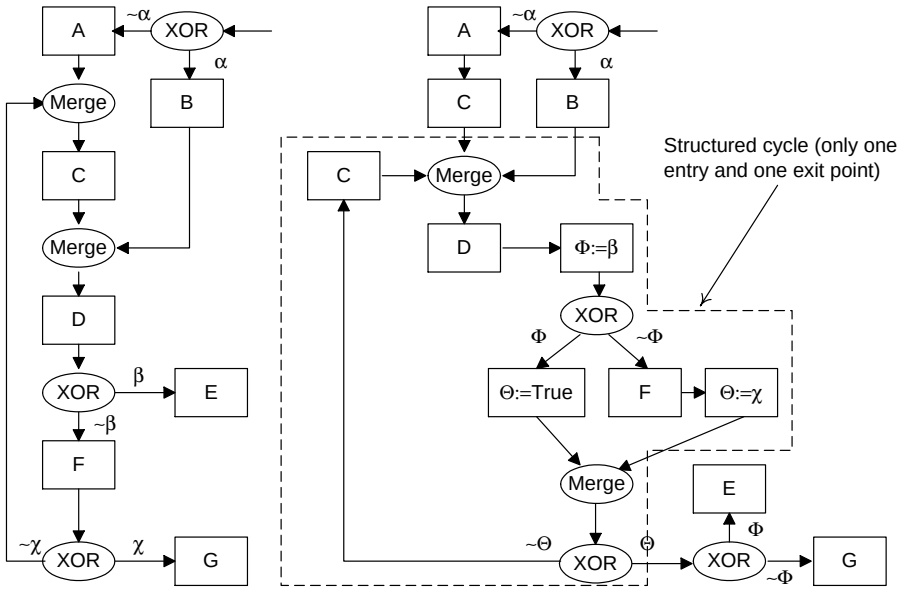

Figure 5: Implementation of arbitrary cycles

Another example of the requirement imposed by some of the workflow engines on a modeler is that the workflow model is to contain only one ending node, or in case of many ending nodes, the workflow model will terminate when the first one is reached. Again, most business models do not follow this pattern - it is more natural to think of a business process as terminated once there is nothing else to be done.

Pattern 12 (Implicit Termination)
Description A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the workflow and no other activity can be made active (and at the same time the workflow is not in deadlock).
Examples

- This semantics is typically assumed for every workflow model at the analysis stage.

Problem Most workflow engines terminate the process when an explicit *Final* node is reached. Any current activities that happen to be running by that time will be aborted. Some workflow engines (Staffware, MQSeries/Workflow, InConcert) would terminate the (sub)process when there is nothing else to be done.
Solutions

- The typical solution to this problem is to transform the model to an equivalent model that has only one terminating node. The complexity of that task depends very much on the actual model. Sometimes it is easy and fairly straightforward, typically by using a combination of different join constructs and activity repetition. However, there are situations where it difficult or even impossible to do so. A model that involves multiple instances (see section 2.4) and implicit termination is typically very hard to convert into a model with explicit termination.

□

2.4 Patterns involving Multiple Instances

The patterns in this subsection involve a phenomenon that we will refer to as *multiple instances*. From a theoretical point of view the concept is relatively simple and corresponds to more than one token in a given place in a Petri-net representation of the workflow graph. From a practical point of view it means that one activity on a workflow graph can have more than one running, active instance at the same time. As we will see, it is a very valid and frequent requirement. The fundamental problem with the implementation of this pattern is that due to design constraints and lack of anticipation for this requirement most of the workflow engines do not allow for more than one instance of the same activity to be active at the same time.

Pattern 13 (Multiple Instances With a Priori Design Time Knowledge)
Description For one case an activity is enabled multiple times. The number of instances of a given activity for a given case is known at design time.
Examples

- The requisition of hazardous material requires three different authorizations.

Problem This pattern typically presents no specific problems.
Solutions

- If the number of instances is known a priori during design time, then a very simple design pattern is to replicate the activity in the workflow model combined with the parallel execution pattern.

□

Pattern 14 (Multiple Instances With a Priori Runtime Knowledge)
Description For one case an activity is enabled multiple times. The number of instances of

a given activity for a given case is variable and may depend on characteristics of the case or availability of resources [CCPP98, JB96], but is known at some stage during runtime, before the instances of that activity have to be created.

Examples

- In the reviewing process of a scientific paper submitted to a journal, the activity *review_paper* is instantiated several times depending on the content of the paper, the availability of referees, and the credentials of the authors.

- For the processing of an order for multiple books, the activity *check_availability* is executed for each individual book.

- When booking a trip, the activity *book_flight* is executed multiple times if the trip involves multiple flights.

- When authorizing requisition with multiple items, each item has to be authorized individually by different workflow users.

Problem Only a few workflow management systems offer a construct for the multiple activation of one activity for a given case. Most systems have to resort to a fixed number of parallel instances of the same activity or an iteration construct where the instances are processed sequentially.

Solutions

- This pattern is a specific instance of Pattern 15 thus any implementation of Pattern 15 will be applicable.

- If there is a maximum number of possible instances, then Patterns 2 and 4 can be used to obtain the desired routing. A XOR-split is used to select the number of instances and triggers one of several AND-splits. For each number of possible instances, there is one AND-split with the corresponding cardinality. The drawback of this solution is that the resulting workflow model can become large and complex and is bound by the maximum number of possible instances.

- Some workflow engines offer a special construct that can be used to instantiate a given number of instances of one activity. An example of such a construct is the *Bundle* concept that is available in MQSeries/Workflow. Once the desired number of instances is obtained (typically by one of the activities in the workflow) it is passed over via the available data flow mechanism to a bundle construct that is responsible for instantiating a given number of instances.

- As in many cases, the desired routing behavior can be supported quite easily by making it more sequential. Simply use iteration (cf. Pattern 11) to activate instances of the activity *sequentially*. Suppose that activity A is followed by n instantiations of B followed by C. First execute A, then execute the first instantiation of B. Each instantiation of B is

followed by a XOR-split to determine whether another instantiation of B is needed or that C is the next step to be executed. This solution fairly straightforward. However, the n instantiations of B are not executed in parallel but in a fixed order. In many situations this is not acceptable. Think of the example of reviewing papers. Clearly, it is not acceptable that the second reviewer has to wait until the first reviewer completes the review, etc.

□

Pattern 15 (Multiple Instances With No a Priori Runtime Knowledge)
Description For one case an activity is enabled multiple times. The number of instances of a given activity for a given case is not known during design time, nor it is known at any stage during runtime, before the instances of that activity have to be created.
Examples

- The requisition of 100 computers involves an unknown number of deliveries. The number of computers per delivery is unknown and therefore the total number of deliveries is not known in advance. Once each delivery is obtained, it can be determined whether a next delivery is to come by comparing the total number of delivered goods so far with the number of the goods requested.

Problem Most workflow engines do not allow more than one instance of the same activity to be active at the same time.
Solutions

- The most straightforward implementation of this pattern is through the use of the loop and the parallel split construct as long as the workflow engine supports multiple instances directly. This is possible in languages such as Forté and Verve.

- Some workflow languages support an extra construct that enables the designer to create a subprocess or a subflow that will "spawn-off" from the main process and will be executed concurrently. For example, Visual WorkFlo supports the *Release* construct while I-Flow supports the *Chained Process Node*.

- If the language does not support a special construct to spawn off the subprocess, then it is typically possible through the API to invoke the subprocess as part of one activity in a process.

- Similarly to Pattern 14, the desired routing behavior can be supported quite easily by making it sequential.

□

The patterns described thus far did not consider the synchronization of multiple instances. For example, spawning off a variable number of subprocess from the main process, as supported

by Visual WorkFlo and I-Flow, does only launch multiple instances without considering synchronization issues.

Pattern 16 (Multiple Instances Requiring Synchronization)
Description For one case an activity is enabled multiple times. The number of instances may not be known at design time. After completing all instances of that activity another activity has to be started.
Examples

- When booking a trip, the activity *book_flight* is executed multiple times if the trip involves multiple flights. Once all bookings are made, the invoice is to be sent to the client.

- The requisition of 100 computers results in a certain number of deliveries. Once all deliveries are processed, the requisition has to be closed.

Problem Most workflow engines do not allow multiple instances. Languages that do allow multiple instances (e.g. Forté, Verve) do not provide any construct that would allow for synchronization of these instances. Languages that support the *Release* construct (Visual WorkFlo, I-Flow) do not provide any construct that would allow for synchronization of spawned off subprocesses.
Solutions

- If the number of instances (or maximum number of instances) is known at design time, then it is easy to synchronize the multiple instances implemented through activity repetition by using basic synchronization.

- If the language supports multiple instances *and* decomposition that does not terminate unless all activities are finished, then multiple instances can be synchronized by placing the workflow sub-flow containing the loop generating the multiple instances inside the decomposition block. The activity to be done once all instances are completed can then follow that block.

- MQSeries/Workflow's *Bundle* construct can be used when the number of instances is known at some point during runtime to synchronize all created instances.

- In most workflow languages none of these solutions can be easily implemented. The typical way to tackle this problem is to use external triggers. Once each instance of an activity is completed, the event should be sent. There should be another activity in the main process waiting for events. This activity will only complete after all events from each instance are received.

□

Figure 6 presents some design patterns for multiple instances. Workflow (a) can be implemented in languages supporting multiple concurrent instances of an activity as well as implicit

termination (see Pattern 12). An activity B will be invoked here many times, activity C is used to determine if more instances of B are needed. Once all instances of B are completed, the subprocess will complete and activity E can be processed. Implicit termination of the subprocess is used as the synchronizing mechanism for the multiple instances of activity B.

Workflow (b) can be implemented in languages that do not support multiple concurrent instances. Activity B is invoked asynchronously, typically through an API. There is no easy way to synchronize all instances of activities B.

Finally workflow (c) demonstrates a simple implementation when it is known during design time that there will be no more than three instances of B.

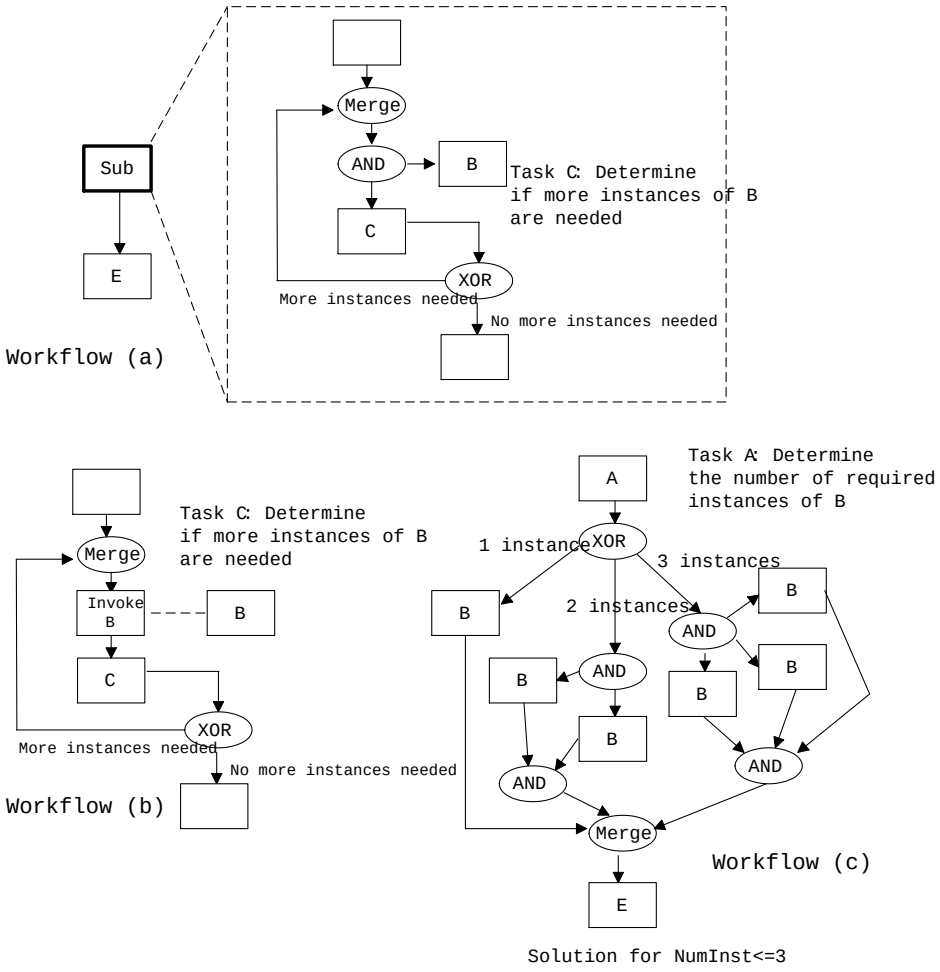

Figure 6: Design patterns for multiple instances

2.5 Temporal Relations

This section describes a pattern, or rather a family of patterns, which refers to the complex temporal relationship between two activities. When describing the most basic workflow pattern, the sequence pattern (Pattern 1) we have made the assumption that the subsequent activity is enabled *after completion* of the preceding one. However, more advanced notions are possible. Let s_1 be the moment the first activity starts and let $e1$ be the moment the activity is completed. Let s_2 and e_2 be the start respectively completion times of the second activity. For the standard sequential routing as described in the sequence pattern, s_2 is later than e_1 ($s_2 \geq e_1$). Other ordering relations similar to the standard sequential routing are (1) before-start ($s_2 \geq s_1$), (2) after-end ($e_2 \geq e_1$), (3) before-end ($e_2 \geq s_1$), and (4) meets ($s_2 = e_1$).

Pattern 17 (Interleaved sequence)
Description Two activities that have an interdependent time relationship other than simple sequencing. It may be that one activity has to start before the second activity finishes or that one activity has to end at exactly the same time as the second activity (but may start at any time).
Synonyms Activity scheduling
Examples

- These patterns are frequently used in the manufacturing domain. With two assembly lines we may want to start both of them at the same time, but the second one should finish the job one hour after the first one finishes its job.

- These patterns are also frequently seen for long-lived activities that can span days or months. Let activity 1 represent an approval process and activity 2 the preparation for the actual construction. We may want to specify that the preparation for the construction can start any time and be executed in parallel with the approval process but it cannot end before the approval process is finished as only then all things are fixed.

Problem Most of the major workflow systems have not been designed to handle these types of applications.
Solutions

- Most workflow systems treat activities as atomic entities. The typical processing of an activity when it is ready to be executed is to route the activity to the appropriate user or actor. The workflow user would then indicate that (s)he has started to work on the activity (which typically involves some locking mechanism on the server) and then (s)he would indicate when the activity is finished. The workflow system is a passive component in this scenario as it is the workflow user that starts and finishes the activity. Common design patterns for providing the required semantics is to split the activity into two parts: begin-activity activity and end-activity activity. Then, depending on the actual workflow product's time management capabilities one can use timers and/or delay nodes

to implement the required semantics. Note that in some cases it may prove to be either very cumbersome or even impossible.

□

2.6 State-based Patterns

In real workflows, most workflow instances are in a state awaiting processing rather than being processed. Most computer scientists, however, seem to have a frame of mind, typically derived from programming, where the notion of state is interpreted in a narrower fashion and is essentially reduced to the concept of data. As this section will illustrate, there are real differences between work processes and computing and there are business scenarios where an explicit notion of state is required. As the notation we have deployed so far is not suitable for capturing states explicitly, we adopt the variant of Petri-nets as described in [Aal98b] when illustrating the patterns in this section. Petri-nets provide a possible solution to modeling states explicitly (examples of commercial workflow management systems based on Petri-nets are COSA and Income).

Moments of choice, such as supported by constructs as XOR-splits/OR-splits, in workflow management systems are typically of an *explicit* nature, i.e. they are based on data or they are captured through decision activities. This means that the choice is made a-priori, i.e. before the actual execution of the selected branch starts an internal choice is made. Sometimes this notion is not appropriate. Consider Figure 7 adopted from [Aal98b]. In this figure two workflows are depicted. In both workflows, the execution of activity A is followed by the execution of B or C. In workflow (a) the moment of choice is as late as possible. After the execution of activity A there is a "race" between activities B and C. If the external message required for activity C (this explains the envelope notation) arrives before someone starts executing activity B (the arrow above activity B indicates it requires human intervention), then C is executed, otherwise B. In workflow (b) the choice for either B or C is fixed after the execution of activity A. If activity B is selected, then the arrival of an external message has no impact. If activity C is selected, then activity B cannot be used to bypass activity C. Hence, it is important to realize that in workflow (a), both activities B and C were, at some stage, simultaneously scheduled. Once an actual choice for one of them was made, the other was disabled. In workflow (b), activities B and C were at no stage scheduled together.

Many workflow management systems abstract from states between subsequent activities, and hence have difficulties modeling implicit choices.

Pattern 18 (Deferred XOR-split)
Description A point in the workflow process where one of several branches is chosen. In contrast to the XOR-split, the choice is not made explicitly (e.g. based on data or a decision) but several alternatives are offered to the environment. However, in contrast to the AND-split, only one of the alternatives is executed. This means that once the environment activates one

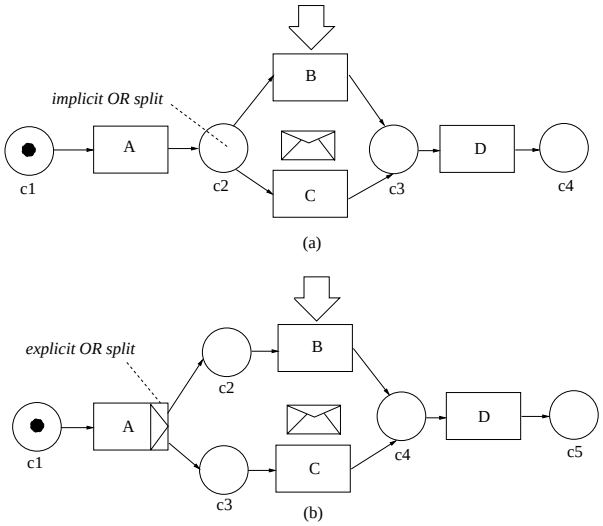

Figure 7: Illustrating the difference between implicit (a) and explicit (b) XOR-splits

of the branches the other alternative branches are withdrawn. It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible.

Synonyms External choice, implicit choice.

Examples

- After receiving the products there are two ways to transport the products to the department. The selection is based on the availability of the corresponding resources. Therefore, the choice is deferred until a resource is available.

- Consider activity A in Figure 7 to represent the activity *send_questionnaire*, and activities B and C, the activities *time_out* and *process_questionnaire*. The activity *time_out* requires a time trigger, while the activity *process_questionnaire* is only to be executed if the complainant returns the form that was sent (hence an external trigger is required for its execution). Clearly, the moment of choice between *process_questionnaire* and *time_out* should be as late as possible. If this choice was modeled as an explicit XOR-split (Pattern 4), it is possible that forms which are returned in time are rejected, or cases are blocked if some of the forms are not returned at all.

Problem Many workflow management systems support the XOR-split described in Pattern 4 but do not support the implicit XOR-split. Since both types of choices are desirable (see example), the absence of the implicit OR-split is a real problem.

Solutions

- Assume that the workflow language being used supports Pattern 2 (AND-split) and Pattern 21 (Cancel activity). The implicit XOR-split can be realized by enabling all

alternatives via an AND-split. Once the processing of one of the alternatives is started, all other alternatives are canceled. Consider the implicit choice between B and C in Figure 7(a). After A, both B and C are enabled. Once B is selected/executed, activity C is canceled. Once C is selected/executed, activity B is canceled. Note that the solution does not always work because B and C can be selected/executed concurrently.

- Another solution to the problem is to replace the implicit XOR-split by an explicit XOR-split, i.e. an additional activity is added. All triggers activating the alternative branches are redirected to the added activity. Assuming that the activity can distinguish between triggers, it can activate the proper branch. Consider the example shown in Figure 7. By introducing a new activity E after A and redirecting triggers from B and C to E, the implicit XOR-split can be replaced by an explicit XOR-split based on the origin of the first trigger. Note that the solution moves part of the routing to the application or task level.

□

Patterns 2 and 3 are typically used to specify parallel routing. Most workflow management systems support true concurrency, i.e. it is possible that two activities are executed for the same case at the same time. If these activities share data or other resources, true concurrency may be impossible or lead to anomalies such as lost updates or deadlocks. Therefore, we introduce the following pattern.

Pattern 19 (Interleaved parallel routing)
Description A set of activities is executed in an arbitrary order: Each activity in the set is executed, the order is decided at run-time, and no two activities are executed at the same moment (i.e. no two activities are active for the same workflow instance at the same time).
Synonyms Unordered sequence.
Examples

- The Navy requires every job applicant has to take two tests: *physical_test* and *mental_test*. These tests can be conducted in any order but not at the same time.

- At the end of each year, a bank executes two activities for each account: *add_interest* and *charge_credit_card_costs*. These activities can be executed in any order. However, since they both update the account, they cannot be executed at the same time.

Problem Since most workflow management systems support true concurrency when using constructs such as the AND-split and AND-join, it is not possible to specify interleaved parallel routing.
Solutions

- A very simple, but unsatisfactory solution, is to fix the order of execution, i.e. instead of using parallel routing, sequential routing is used. Since the activities can be executed in

an arbitrary order, a solution using a predefined fix order may be acceptable. However, by fixing the order, flexibility is reduced and the resources cannot be utilized to their full potential.

- Another solution is to use a mixture of Patterns 4 and 1, i.e. several alternative sequences are defined and before execution one sequence is selected using a XOR-split. A drawback is that the order is fixed before the execution starts. Moreover, the workflow model may become quite complex and large by enumerating all possible sequences.

- By using Pattern 18 (instead of Pattern 4) the order does not need to be fixed before the execution starts, i.e. the implicit OR-split allows for on-the-fly selection of the order. Unfortunately, the resulting model typically has a "spaghetti-like" structure.

- For workflow models based on Petri nets, the interleaving of activities can be enforced by adding a place which is both an input and output place of all potentially concurrent activities. The AND-split adds a token to this place and the AND-join removes the token. It is easy to see that such a place realizes the required "mutual exclusion". See Figure 8 for an example where this construct is applied. Note that, unlike the other solutions, the structure of the model is not compromised.

□

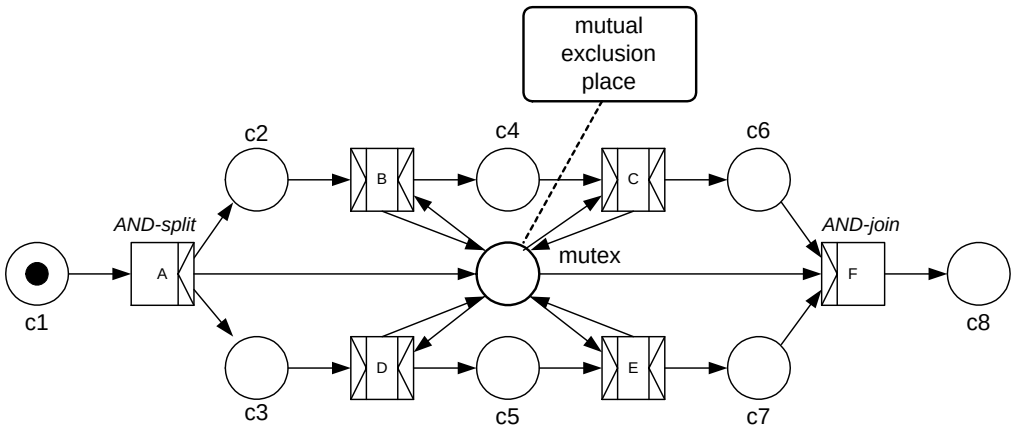

Figure 8: The execution of B, C, D, and E is interleaved by adding a mutual-exclusion place.

The expressive power of many workflow management systems is restricted by the fact that they abstract from states, i.e. the state of a workflow instance is not modeled explicitly. The solution shown in Figure 8 is only possible because mutual exclusion can be enforced by place *mutex* (i.e. state information shared among the activities). Pattern 18 (implicit XOR-split) is another example of a construct which is hard to handle if one abstracts from the states in-between activities. The next pattern, Pattern 20, allows for testing whether a case has reached

a certain phase. By explicitly modeling the states in-between activities this pattern is easy to support. However, if one abstracts from states, then it is hard, if not impossible, to test whether a case is in a specific phase.

Example 2.1 Consider the workflow process for handling complaints (see Figure 9). First the complaint is registered (activity *register*), then in parallel a questionnaire is sent to the complainant (activity *send_questionnaire*) and the complaint is evaluated (activity *evaluate*). If the complainant returns the questionnaire within two weeks, the activity *process_questionnaire* is executed. If the questionnaire is not returned within two weeks, the result of the questionnaire is discarded (activity *time_out*). Based on the result of the evaluation, the complaint is processed or not. The actual processing of the complaint (activity *process_complaint*) is delayed until the questionnaire is processed or a time-out has occurred. The processing of the complaint is checked via activity *check_processing*. Finally, activity *archive* is executed. □

The construct involving activities *process_complaint* which is only enabled if place *c4* contains a token is called a milestone.

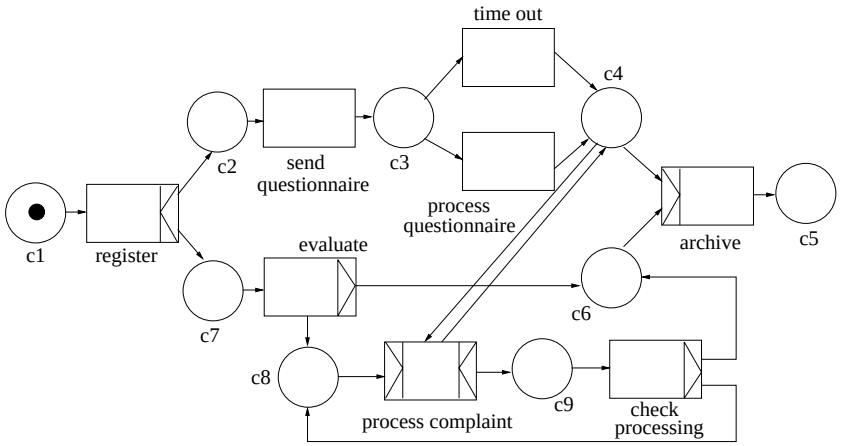

Figure 9: The state in-between the processing/time-out of the questionnaire and archiving the complaint (i.e. place *c4*) is an example of a milestone.

Pattern 20 (Milestone)
Description The enabling of an activity depends on the case being in a specified state, i.e. the activity is only enabled if a certain milestone has been reached which did not expire yet. Consider three activities A, B, and C. Activity A is only enabled if activity B has been executed and C has not been executed yet, i.e. A is not enabled before the execution B and A is not enabled after the execution C.
Synonyms Test arc, deadline (cf. [JB96]), state condition.
Examples

- In a travel agency, flights, rental cars, and hotels may be booked as long as the invoice is not printed.

- A customer can withdraw purchase orders until two days before the planned delivery.

- A customer can claim air miles until six months after the flight.

- The construct involving activity *process_complaint* and *c4* shown in Figure 9.

Problem The problem is similar to the problem mentioned in Pattern 18: There is a race between a number of activities and the execution of some activities may disable others. Note that in Figure 9 activity *process_complaint* may be executed an arbitrary number of times, i.e. it is possible to bypass *process_complaint*, but it is also possible to execute *process_complaint* several times.

Solutions

- Consider three activities A, B, and C. Activity A can be executed an arbitrary number of times before the execution of C and after the execution of B. Such a milestone can be realized using Pattern 18. After executing B there is an implicit XOR-split with two possible subsequent activities: B and C. If B is executed, then the same implicit XOR-split is activated again. If C is executed, B is disabled by the implicit XOR-split construct. Note that this solution only works if the execution of B is not restricted by other parallel threads. For example, the construct cannot be used to deal with the situation modeled in Figure 9 because *process_complaint* can only be executed directly after a positive evaluation or a negative check, i.e. the execution of *process_complaint* is restricted by both parallel threads.

- Another solution is to use the data perspective, e.g. introduce a Boolean workflow variable m. Again consider three activities A, B, and C such that activity A is allowed to be executed in-between B and C. Initially, m is set to false. After execution of B m is set to true, and activity C sets m to false. Activity A is preceded by a loop which periodically checks whether m is true: If m is true, then A is activated and if m is false, then check again after a specified period, etc. Note that this way a "busy wait" is introduced. More sophisticated variants of this solution are possible by using database triggers, etc. However, a drawback of this solution approach is that an essential part of the process perspective is hidden inside activities and applications. Moreover, the mixture of parallelism and choice may lead to all kinds of concurrency problems.

□

Having introduced the milestone pattern another solution to Pattern 16 can be given (see Figure 10). This solution uses a mixture of a loop construct to enable multiple instances of B in parallel, and a milestone and XOR-split to detect the completion of all instances. Note that the enabling is done sequentially, the actual execution, however, is done in parallel. The

AND-split X enables one instance of B and activates Y. The XOR-split Y checks whether all instances have been enabled. The AND-join/OR-split Z uses a milestone-like construct and is activated for every instantiation of B. Z determines whether all instances have been processed, i.e. it waits for the next completion of an instance of B or enables C.

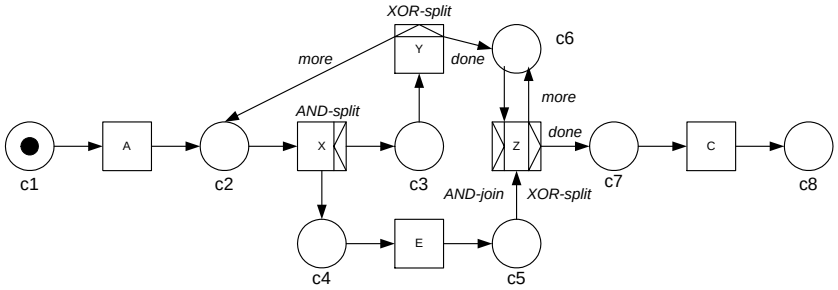

Figure 10: Multiple activation of activity B using the milestone pattern

It is interesting to think about the reason why many workflow products have problems dealing with patterns 18, 19, and 20. The systems that abstract from states are typically based on messaging, i.e. if an activity completes, it notifies or triggers other activities. This means that activities are *enabled* by the receipt of one or more messages. Patterns 18, 19, and 20 have in common that an activity can become *disabled* (temporarily). However, since states are implicit and there are no means to disable activities (i.e. negative messages), these systems have problems dealing with the constructs mentioned. Note that the synchronous nature of patterns 18, 19, and 20 further complicates the use of asynchronous communication mechanisms such as message passing using "negative messages" (e.g. messages to cancel activities).

2.7 Cancellation Patterns

In this section focus is on patterns dealing with cancellation of activities and cases.

The first solution described in the Deferred XOR-split (Pattern 18) uses a construct where one activity cancels another, i.e. after the execution of activity B, activity C is withdrawn and after the execution of activity C activity B is withdrawn. The following pattern describes this construct.

Pattern 21 (Cancel activity)
Description An enabled activity is disabled, i.e. a thread waiting for the execution of an activity is removed.
Synonyms Withdraw activity.
Examples

- Normally, a design is checked by a two groups of engineers. However, to meet deadlines it is possible that one of these checks is withdrawn to be able to meet a deadline.

- If a customer cancels a request of information, the corresponding activity is disabled.

Problem Only a few workflow management systems support the withdrawal of an activity directly in the workflow modeling language, i.e. in a (semi-)graphical manner.
Solutions

- If the workflow language supports Pattern 18 (implicit OR-split), then it is possible to cancel a activity by adding a so-called "shadow activity". Both the real activity and the shadow activity are preceded by an implicit OR-split. Moreover, the shadow activity requires no human interaction and is triggered by the signal to cancel the activity. Consider for example a workflow language based on Petri nets. An activity is canceled by removing tokens from its input place. The tokens are removed by executing another activity having the same set of input places. Note that the drawback of this solution is the introduction of activities which do not correspond to actual steps of the process.

- Many workflow management systems support the withdrawal of activities using an API which simply removes the corresponding entry from the database, i.e. it is not possible to model the cancellation of activities in a direct and graphical manner, but inside activities one can initiate a function which disables another activity.

□

A similar construct is the cancellation of an entire case.

Pattern 22 (Cancel case)
Description A case, i.e. workflow instance, is removed completely.
Synonyms Withdraw case.
Examples

- In the process for hiring new employees, an applicant withdraws his/her application.

- A customer withdraws an insurance claim before the final decision is made.

Problem Workflow management systems typically do not support the withdrawal of an entire case using the (graphical) workflow language.
Solutions

- Pattern 21 can be repeated for every activity in the workflow process definition. There is one activity triggering the withdrawal of each activity in the workflow. Note that this solution is not very elegant since the "normal control-flow" is intertwined with all kinds of connections solely introduced for removing the workflow instance.

- Similar to Pattern 21, many workflow management systems support the withdrawal of cases using an API which simply removes the corresponding entry from the database.

□

2.8 Inter-Workflow Synchronization

The previous patterns mainly related to triggering dependencies within a single workflow. In this section, we turn our attention to triggering dependencies across workflows, specifically the synchronization of activities in different workflows. The requirement typically emerges from business-to-business (B2B) interaction where a number of agencies collaborate for the provision of a business service. Each agency has a separate workflow and may not want its details to be exposed (e.g. for competitive purposes). However, some activities in a workflow will warrant information produced from another workflow.

Different configurations can be used for workflow interoperability. One is having separate workflows run by separate and possibly heterogeneous workflow systems which can communicate with each other. Another is to have separate workflows run on a single workflow system, which removes the problem of requiring heterogeneous workflow systems having to communicate. Given some open issues with heterogeneous workflow systems interoperability, our focus will be on a single workflow system configuration and the essential issues of inter-workflow synchronization therein.

For the purposes of what we describe as inter-workflow *messaging*, we define two fundamental messaging activities, a message sender and a message receiver. The execution of a *message sender* involves the transmission of a message within a specified scope. Given the control-flow perspective of this paper, we ignore the specification of the message itself, except to say that a typing mechanism would be employed to define permissible instances of messages for senders and receivers, including passing no/null data - analogous to transmitting a signal. The explicit specification of the *message scope* at the workflow level serves to identify the destination of the message send without having to prescribe the message receiver. Figure 11 illustrates a specification for sender A in one workflow X, receiver B in another workflow Y, and their communication through message scope m. Communication can take place since both sender and receiver share the same message scope.

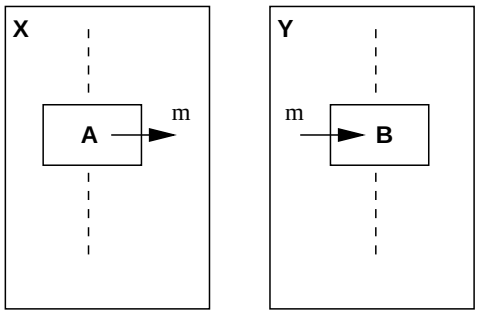

Figure 11: Messaging at the workflow level

Pattern 23 (Messaging communication)
Description The transmission of a message from a message sender in one workflow (or

compound activity) to a message receiver in another workflow. The message scope of the sender and receiver must be the same.

At run-time, communication can only occur between one sender and one receiver instance. If one sender and multiple receivers exist in the same scope, then only one potential communication can take place, and the non-matching receiver instances will be deadlocked. If several senders but one receiver exist, one communication will take place but no deadlocks will be incurred. This is because receivers only involve a wait dependency. The message receive can be seen as an AND-join (of the local triggers and an external messaging trigger).

A further variant on message sending is synchronicity, i.e. where a sender will not complete until it "knows" that its message has been received. Alternatively, a receiver could follow a sender to emulate synchronous sending. We favor a dedicated treatment for synchronous sending to reduce the imperative burden on designers. The issue of synchronous sending will not be discussed further.

Synonyms Event produce-consume.

Examples

- During the processing of particular types of insurance claims, insurance companies collaborate with damage assessment agents. When an insurance processing workflow instance is started, a workflow for a selected damage assessment agent is also triggered to perform some preparation and to then wait for a *damage_assessment_notification* from the insurance company workflow. The damage assessment agents do not want insurance companies to "see" what activities they undertake (e.g. which searching and verification agents they themselves canvass). To prevent this, a *damage_assessment_notification* is the message scope within which a sender activity in the insurance company workflow communicates with a receiver activity in the damage assessment agent workflow.

Problem Most workflow management systems provide the ingredients for messaging at the implementation level, namely through application handlers which can be written to send and receive messages using the workflow management system's messaging middleware service. The problem is that the receiver task should be enabled (by the workflow engine) during the time that the message is in internal transport. This will not always be the case for all types of messaging middleware technologies (as message queues are finite in size and as new messages come into the queue, older ones will go out). Thus, decoupling messaging from the workflow definition level and reliance solely on the implementation messaging services will not guarantee that the communication pattern will work.

Solutions

- The solution is to provide messaging at the workflow definition such that the workflow management system ensures that communication can occur. Specifically, workflow managers should ensure that the messages persist, at least during the time that their potential communications can take place. There should no loss of workflow engine's activity scheduling in the presence of message sending, waiting and receipt.

Such a solution with different approaches can be seen in different products. As an example the SAP R/3 Workflow's so-called event-process chains are based on *event* types at the start and end of workflow activities, where events have an explicit definition. Thus, events of the same type can be used to define the message scope. The event type would be triggered by a message sender, while the same event type would trigger a message receiver. Events persist as objects in the database of SAP R/3 installation.

Another example is FileNet Visual Workflo which persists events associated with the creation of (work object) activities, allowing these to pre-condition the execution of other (work object) activities. This event type is called WaitCreate. Thus, the communication pattern would be achieved through a WaitCreate event on a waited activity, i.e. a message receiver, and a waited-on activity, i.e. a message sender. Encapsulation is not broken as the actual message sender name does not have to be revealed in the WaitCreate event - merely a symbolic name associated with the message sender. This, of course, would capture the message scope.

It is worth noting, too, that proposals are underway for the standardization of workflow level event handling, allowing the communication pattern, as seen through IBM, Sun and DSTC's joint submission for the Object Management Group's (OMG) enterprise process modeling standard[1]

□

Pattern 24 (Messaging coordination)
Description A sender issues a request and at the "sending" end a response is anticipated for that request by a subsequent receiver associated with the sender. Figure 12 illustrates how coordinated messaging applies.

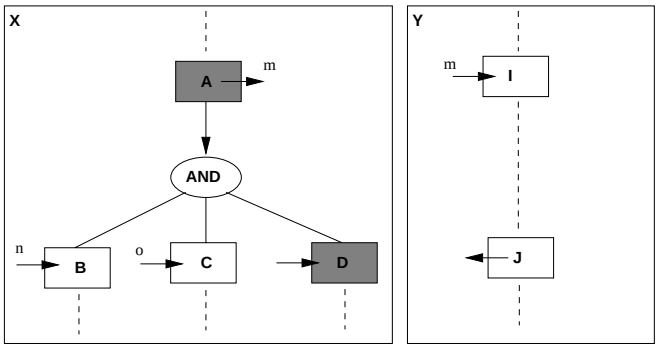

Figure 12: Messaging coordination

[1]This submission is part of the Enterprise Distributed Object Computing (EDOC) profile for the Unified Modeling language (UML) standardization http://www.omg.org/techprocess/meetings/schedule/-UML_Profile_for_EDOC_RFP.html which is currently in progress.

At the "sending" end X, a sender A issues a request through a message scope m. In the sender's execution path, a receiver D, associated with the sender, is activated to receive a returning response from an outside activity which relates to the sender's request - e.g. the response required is an update of the same notification sent in the request. At the "receiving" end Y, a receiver I communicates with sender A in message scope m, and in a subsequent activity, a sender J associated with I, sends back a response to receiver D, associated with the originator of the request, A.

From a "sending" point of view X, messaging coordination is characterized by a sender, the sender's message scope and an associated receiver. Coordination can similarly be described from a "receiving" point of view Y.

Examples

- Continuing the insurance example from the message communication pattern (above), consider an extension to the interaction where the insurance company sends out a request for a *damage_assessment_notice* while continuing processing. As part of further processing, the insurance company awaits possible further *claim_details* or *cancellations* from the claimer as well as the *damage_assessment_notice*. For *claim_details* and *cancellations*, uncoordinated receiver activities can be used. However, for the *damage_assessment_notice* a coordination relationship is required between the sender of the request (notice) and the receiver of the response (the update of the same notice).

Problem Two-way communication (using pattern 23 from the "sending" to the "receiving" end, and from the "receiving" end to the "sending" end) could be used to achieve the coordination pattern. Thus, workflow systems achieving the communication pattern could also achieve the coordination pattern. However, there is the additional requirement of associating the sender and its anticipating receiver in the same workflow. A difficulty, in this regard, arises from the existence of multiple instances of the same sender and receiver types concurrently (through e.g. cyclic iteration), meaning sender and receiver instances need to be associated at runtime. Otherwise, for example, a receiver instance might obtain an update of a notification originally requested through a sender instance which did not trigger it.

Solutions

- The particular problem of dynamic association described above can be achieved by making the instance identifier of the sender available to the anticipating receiver. This same instance identifier would also be passed through communication from the "sending" to the "receiving" end, and then back to the "sending" end, where it would have to match up with the instance identifier passed to the anticipating receiver. This instance identifier must be available prior to the arrival of the response.

  As an example, in SAP R/3 Workflow and FileNet Visual Workflo, the dynamic association of sender and receiver can be achieved through an event trigger between the sender and receiver; the enabling of the receiver depends on the sender's prior creation and the instance identifier of the sender is passed onto the receiver through the event trigger on the receiver. Note in SAP R/3 Workflow, multiple instances of the same type cannot

exist concurrently, so there is no strict need for dynamic association. Nevertheless, the event trigger solution is general enough for static and dynamic associations.

□

Pattern 25 (Bulk message sending)
Description Multiple instances of message senders of the same type execute concurrently. This allows the capture of business situations where notifications of the same type are sent to several, external stakeholders. The number may be known a priori at design time or runtime, or may only be determined during runtime.
Synonyms Multi-cast messaging.
Examples

- An online share broker enables trading transactions depending on a customer's buying and selling needs. A *trade* is made if *offers* are accepted for a customer's buying and selling *shares*. *Offers* are raised and negotiated (through several cycles) using message senders. For different cycles, the number might be known a priori at runtime or may vary during runtime, e.g. depending on the availability of buying shares. The share market, being highly dynamic and distributed, requires as much parallelism as possible for the messaging associated with making and accepting *offers*.

Problem Multiple instances of the same message sender type are required concurrently. Because multiple, concurrent instances are not widely supported in current workflow management systems, more imperative solutions like the cyclic iteration of a message sender are adopted. This, however, involves sequential not concurrent message sending, which may be unsuitable for certain requirements where, for instance, there may be a legal requirement to send notifications at the same time.
Solutions

- A message sender activity is applied through the one of the multiple instances patterns, depending on whether the number is known a priori at design time (Pattern 13), known a priori at runtime (Pattern 14) or with no a priori runtime knowledge (Pattern 15). FileNet Visual Workflo supports the communication pattern, and therefore message sending, as well as the multiple instances patterns. It therefore achieves bulk sending.

□

Pattern 26 (Bulk message receiving)
Description Multiple instances of message receivers of the same type execute concurrently. This allows the capture of business situations where the notifications are received from several, external stakeholders. The number may be known a priori at design time or runtime, or may only be determined during runtime.
Examples

- Continuing with the example of the online share broker (described in Pattern 25), for offers made through a bulk message sender, responses should be received. The maximum number of possible incoming responses is known at runtime, namely the same the number of associated message senders. (Note, the implicit requirement of coordination required for senders and receivers).

Problem As with bulk message sending, the lack of support for multiple, concurrent instances can lead to a "sequentialization" of message receiving. Doing this for message receiving is a graver problem since it is clearly suboptimal reception - only one message is received at a time.
Solutions

- A message receiver activity is applied through the multiple instances patterns, where the number is known a priori at design time (Pattern 13), known a priori at runtime (Pattern 14) or with no a priori runtime knowledge (Pattern 15). As with bulk messaging, only FileNet Visual Workflow supports the communication pattern, and therefore message receiving as well as the multiple instances patterns. It therefore achieves bulk receiving.

□

3 Comparing Workflow Management Systems

3.1 Introduction

The workflow patterns described in this paper correspond to routing constructs encountered when modeling and analyzing workflows. Many of the patterns are supported by workflow management systems. However, several patterns are difficult, if not impossible, to realize using many of the workflow management systems available today. As indicated in the introduction, the routing functionality is hardly taken into account when comparing/evaluating workflow management systems. The system is checked for the presence of sequential, parallel, conditional, and iterative routing without considering the ability to handle the more subtle workflow patterns described in this paper. The evaluation reports provided by prestigious consulting companies such as the "Big Six" (Andersen Worldwide, Ernst & Young, Deloitte & Touche, Coopers & Lybrand, KPMG, and Price Waterhouse) typically focus on purely technical issues (Which database management systems are supported?), the profile of the software supplier (Will the vendor be taken over in the near future?), and the marketing strategy (Does the product specifically target the telecommunications industry?). As a result, many enterprises select a workflow management system that does not fit their needs.

In this section, we provide a comparison of the functionality of 12 workflow management systems (COSA, Visual Workflow, Forté Conductor, Meteor, Mobile, MQSeries/Workflow, Staffware, Verve Workflow, I-Flow, InConcert, Changengine, and SAP R/3 Workflow) based on the (advanced) workflow patterns presented in this paper (the elementary patterns are

omitted in the comparison as they can be realized by all WFMSs). We would like to stress that the product-specific information at our disposal was current at the end of 1999 and we cannot guarantee that all the results are still valid.

3.2 Products

Before we compare the products based on the workflow patterns presented in this paper, we briefly introduce each product and supply some background information.

COSA [SL96] is a Petri-net-based workflow management system developed by Ley GmbH (formerly operating under the names Software Ley and COSA Solutions). Ley GmbH is a German company based in Pullheim (Germany) and is part of the Baan Consortium (i.e. a member of the Vanenburg Group). COSA is one of the leading workflow management systems in Europe and can be used as a stand-alone workflow system or as the workflow module of the Baan IV ERP system. COSA will also be used as the workflow engine of the new BaanSeries platform. For our evaluation we used version 2.0. The modeling language of COSA consists of two types of building blocks: activities (i.e., Petri net transitions) and conditions (i.e. Petri net places). COSA extends the classical Petri net model with control data to allow for explicit choices based on information and decisions. Unfortunately, only safe Petri nets are allowed, i.e., it is not allowed to have multiple tokens in one place. Therefore, COSA is unable to support multiple instances directly. The only way to deal with multiple instances is to use workflow triggers. Every subprocess in COSA has a unique start activity and a unique end activity. As a result, only highly structured subprocesses are possible and termination is always explicit. The main feature of the workflow language of COSA is that it allows for the explicit representation of states. As a result, state-based patterns such as the Deferred XOR-split, and Interleaved parallel routing are supported in a direct and graphical manner. Tasks can be removed from places, providing support for Cancel Activity, however COSA does not have an explicit provision for Cancel Case other than through its API. Messaging Communication and Coordination are supported in COSA through essentially external triggers which can be input to, or output from, activities. Since Multiple Instances are not supported, COSA does not support the bulk messaging patterns.

Visual WorkFlo is one of the market leaders in the workflow industry. It is part of the FileNet's Panagon suite that includes also document management and imaging servers. Visual WorkFlo is one of the oldest and best established products on the market. Since its introduction in 1994 it managed to gain a respectable share of all worldwide workflow applications. FileNet as a corporation ranks amongst the top 60 software companies in the world (Software magazine) - with offices in 13 countries and over 650 Value Added Resellers building solutions on top of Panagon's suite. The workflow modeling language of Visual WorkFlo is highly structural and is a collection of activities and routing elements such as Branch (XOR-split), While (structured loop), Static Split (AND-split), Rendevous (AND-join), and Release. Visual WorkFlo does not directly support any of the advanced synchronization patterns. It requires

the model to have structured loops only and one, explicit, termination node thus limiting the suitability of the resulting specifications. Direct support for Multiple Instances is possible through the *Release* construct as long as there is no further synchronization required. There is no direct way to implement any of the state-based patterns. There is no explicit support for the cancellation patterns. However, Visual Workflo supports all the messaging patterns: Messaging Communication and Coordination are possible through event (WaitCreate) dependency of one activity by another, with the support of Multiple Instances also making bulk messaging possible.

Forté Conductor is a workflow engine that is an add-on to Forté's powerful development environment, Forté 4GL (formerly Forté Application Environment). Forté Software has recently (in October 1999) been acquired by Sun Microsystems. Conductor's engine is based on experimental work performed at Digital Research and its modeling language is powerful and flexible. The workflow model in Conductor comprises a set of activities connected with transitions (called *Routers*). Each transition has associated transition conditions. Each activity has a trigger that determines the semantics of that activity if it has more than one incoming transition. The triggers are flexible enough for easy specification of OR-join, AND-join and any type of N-out-of-M join (see Pattern 10) although the semantics of such a specification is implicit and not visible to the end-user. Arbitrary cycles are supported, but explicit termination points are required. Forté supports creation of multiple instances directly (through the use of a multi-merge join) but does not support any direct means of their further synchronization. State-based patterns cannot be realized. Forté does not have a construct for Cancel Activity but Cancel Case is available through its termination semantics - when an activity is executed which has no other triggers, it will terminate that workflow decomposition. Forté does not support the messaging patterns.

Meteor (Managing End-To-End OpeRations) [SKM] is a CORBA-based workflow management system developed by members of the LSDIS laboratory of the University of Georgia (USA). Interesting features of Meteor are the support for transactional workflows and the full exploitation of Web, CORBA, and Java based distributed computing infrastructures. The Meteor project is funded through the NIST ATP initiative in Information Infrastructure for Healthcare and involves 17 IT and healthcare institutions. Meteor has been tested by several industry partners and is in the process of being commercialized by Infocosm Inc. A workflow in Meteor is defined as a collection of activities and dependencies. An activity can be any combination of AND/XOR-joins and AND/XOR-splits and there are two types of dependencies: control dependencies and data dependencies. The focus of Meteor is on transactional features and distribution aspects. The workflow modeling language supports few of the more advanced constructs. For example, it is not possible to handle any of the state-based patterns, multiple instances are not supported explicitly, termination is always explicit, and the Synchronization merge, Discriminator and cancellation are not supported. Of the messaging patterns, Communication and Coordination are supported. The Multi-merge and Arbitrary cycles patterns are supported. The N-out-of-M pattern is only possible by listing all possible combinations.

Mobile [JB96] is a workflow management system developed by members of the Database Systems group at the University of Erlangen/Nürnberg (Germany). It is a research prototype with several interesting features, e.g. the system is based on the observation that a workflow comprises many perspectives (cf. [JB96]) and one can reuse each perspective separately. The control-flow perspective of Mobile offers various routing constructs to link so-called "workflow types". A workflow type is either an elementary activity or the composition of other workflow types. A powerful feature of the Mobile language is that the set of control-flow constructs is not fixed, i.e. the language is extensible. It is possible to add any of the design patterns identified in this paper as a construct. To add a construct, one can use the Mobile editor MoMo to add the graphical representation of the construct. The semantics is expressed in terms of Java. Since the Java code has direct access to the state of the workflow instance, all routing constructs can be supported. The fact that the language is extensible makes the workflow language of Mobile hard to compare with the other languages. To make a fair comparison we only considered the routing constructs currently available in Mobile. The standard constructs of Mobile include, in addition to the basic patterns, the N-out-of-M join and interleaved parallel routing.

MQSeries/Workflow is the successor of IBM's major workflow offering, FlowMark. FlowMark was one of the first workflow products that was independent from document management and imaging services. It has been renamed to MQSeries/Workflow after a major move from the proprietary middleware to middleware based on the MQSeries product. The workflow model consists of activities linked by transitions. Other than a decomposition block, few other special modeling constructs available. The workflow engine of MQSeries/Workflow has a unique execution semantics in that it propagates a *False Token* for every transition with a condition evaluating to False. This allows for every activity that has more than one incoming transition to act as a synchronizing merge (see Pattern 7). Other than the synchronizing merge, which is a natural construct for MQSeries/Workflow, there is no way to directly implement any of the other advanced synchronization patterns. Support for multiple instances is provided through the *Bundle* construct although it is not suitable if the number of instances is not known at any point prior to generating the instances involved. Arbitrary loops are not supported. An explicit termination point is not required and the workflow process will terminate when "there is nothing else to be executed". There is no direct way to model the state-based, cancellation, and messaging patterns.

Staffware [Sta97] is one of the leading workflow management systems. Staffware is authored and distributed by Staffware PLC. Staffware PLC has its headquarteris in Maidenhead (UK), operates through offices in 15 countries and has a network of 360 partners, resellers and OEMs. We used both the most recent version of Staffware (i.e. Staffware 2000), which was released in the last quarter of 1999, and Staffware 97 [Sta97] for our evaluation. This latter version of Staffware is used by more than 550,000 users worldwide and runs on more than 4500 servers. In 1998, it was estimated by the Gartner Group that Staffware has 25 percent of the global market [Cas98]. The routing elements used by Staffware are the Start, Step, Wait, Condition, and Stop. The Step corresponds to an activity which has an OR-join/AND-split semantics. The Wait step is used to synchronize flows (i.e. an AND-join) and conditions are used for conditional

routing (i.e. XOR-split). Arbitrary loops are supported. There is no direct provision for multiple instances nor for the advanced synchronization constructs. There is no need to define explicit termination points, i.e. termination is implicit. Staffware does not offer a state concept. The so-called "withdraw" transition allows the Cancel Activity pattern to be supported. No support is available for Cancel Case, nor are the messaging patterns supported.

Verve is a relative newcomer to the workflow market. What makes it an interesting workflow product is that it has been designed from the ground up as an embeddable workflow engine. The workflow engine of Verve is very powerful and amongst other features allows for multiple instances and dynamic modification of running instances. The Verve workflow model consists of activities connected by transitions. Each transition has an associated transition condition. Extra routing constructs such as synchronizer and discriminator are supported. Arbitrary loops are supported. An explicit termination point is required. Multiple instances are directly supported (through the use of the multi-merge) as long as they do not require subsequent synchronization. There is no direct way to implement state-based patterns. Of the cancellation patterns, Cancel Case is supported through the forced termination by the "first of the last" activities which terminates. None of the messaging patterns are supported directly, although with Verve the provision of source code and its open, embeddable design make it possible for sites to implement additional constructs including message/event handling from the "ground-up" as opposed to implementing workarounds.

I-Flow is a workflow offering from Fujitsu that can be seen as a successor of the well-established workflow engine from the same company, TeamWare. I-Flow is web-centric and has a Java/CORBA based engine built specifically for Independent Software Vendors and System Integrators. The workflow model in I-Flow consists of activities and a set of routing constructs connected by transitions (called *Arrows*). Routing constructs include Conditional Node (XOR-split), OR-NODE (Merge), and AND-NODE (synchronizer). The AND-split can be modeled implicitly by providing an activity with more than one outgoing transition. Multiple instances can be implemented using the *Chained Process Node* which allows for asynchronous subprocess invocation. Arbitrary loops are allowed but the process requires an explicit termination point. There is no direct way to implement state-based patterns. Cancel Case but not Cancel Activity is supported. None of the messaging patterns are supported.

InConcert has been established in 1996 as a Xerox fully-owned subsidiary. In 1999 it has been bought by TIBCO Software. InConcert 2000 is the newest version of their flagship workflow offering. An InConcert workflow definition is called a "job". A job can contain none, one or many activities. An activity is either simple or compound. An activity can be connected to an arbitrary number of other activities but circular dependencies are not allowed. Each activity has a perform condition attached to it. The default setting of the perform condition is "true" such that activities can be executed in general. If the perform condition evaluates to "false", the activity is skipped. If an activity is skipped, then the subsequent activities are not skipped automatically. Conditional branching or case branching can be achieved by parallel activities with different perform conditions. Arbitrary cycles are not supported. An explicit termination point is not required. There is no direct provision for multiple instances nor for

direct implementation of the state-based patterns. The cancellation and messaging patterns are not supported.

SAP R/3 Workflow[2] SAP is the main player in the market of ERP systems. Its R/3 software suite includes an integrated workflow component that we have evaluated independently of the rest of R/3. SAP workflow models are designed using so-called Event-driven Process Chains (EPC), which consist of a set of functions (activities), events and connectors (AND, XOR, OR). However, in SAP R/3 Workflow not the full expressive power of EPCs can be used, as there are a number of syntactic restrictions similar in vein to the restrictions imposed by Filenet Visual Workflo (e.g. every workflow needs to have a unique starting and a unique ending point, and-splits are always followed by and-joins, or-splits by or-joins etc). As such, there is no direct provision for the advanced synchronization constructs (with one exception: it is possible to specify for the join operator how many parallel branches it has to wait for, hence its semantics corresponds to the N-out-of-M join), multiple instances, arbitrary loops, state-based or cancellation patterns. The event constructs which can trigger or be triggered by activities, which are directly supported at the workflow level, allow support for the Messaging Communication and Coordination patterns. However, the lack of support for Multiple Instances means that bulk messaging cannot be supported.

Changengine is a workflow offering from HP, the second largest computer supplier in the world. Version 3.0 of the product has been introduced in 1998 and it is focused on high performance and support for dynamic modifications. Workflow models in Changengine consist of a set of work nodes and routers linked by arcs. A work node can have only one incoming and one outgoing arc. If more transitions are required, they have to be created explicitly through the router node. Router node semantics is determined by the set of *route rules*. Arbitrary loops are allowed. Changengine does not provide any support for multiple instances. The termination policy is rather unusual: the process will terminate once all process nodes without outgoing activities (*End Points*) are reached. There is no direct way to implement the state-based patterns. A routing rule associated with an activity can be set to cause termination of a decomposition, thus supporting Cancel Case. The Cancel Activity pattern is not supported, nor are the messaging patterns.

3.3 Results

Tables 1 and 2 summarize the results of the comparison of the workflow management systems in terms of the selected patterns. For each product-pattern combination, we checked whether it is possible to realize the workflow pattern with the tool. As each pattern is different, it is hard to come up with a characterization that would fit all of them. For that reason we are summarizing the evaluation criteria for each pattern:

- **Synchronization Merge, Multi-Merge, Discriminator, N-out-of-M join**. If the workflow management system supports the construct that can be used for implementing

[2]The documentation of release 3.1, May 1997, was used as source.

these patterns, it is rated +, otherwise it is rated -. Changengine and Forté receive an intermediate rating as they can realize these patterns only through requiring analysts to explicitly specify corresponding rules. In Meteor the N-out-of-M join is possible by listing all possible combinations. Therefore, an intermediate rating is given.

- **Arbitrary Loops**. If the workflow management system allows for the specification of arbitrary loops, it is ranked + otherwise it is ranked -.

- **Implicit termination**. If the workflow management system requires the specification of an explicit termination point, it is rated -, otherwise it is rated +.

- **Multiple Instances**. If a workflow management system supports the specification that will result in multiple instances without resorting to the use APIs, it is rated +, otherwise it is rated -. MQSeries/Workflow receives an intermediate rating as through its Bundle concept it supports multiple instances where the number of instances is known a priori.

- **State-based patterns**. If the workflow management system supports the direct implementation of the state-based patterns, it is rated +, otherwise it is rated -.

- **Cancellation**. If a workflow management system provides the specification of cancellation of activities/cases without resorting to APIs, it is rated +, otherwise it is rated -.

- **Messaging**. If a workflow management system supports the specification of message/event handling without resorting to APIs, it is rated +, otherwise it is rated -. Message/event handling outside a workflow management system is not evaluated as there are open issues associated with the interoperability of workflow management systems. Note, the bulk messaging patterns rated a + if the workflow management system supported Multiple Instances. Verve receives an intermediate rating as it provides an open embeddable configuration which allows "ground-up" implementations which are not API workarounds.

From the comparison it is clear that no tools support all the selected patterns. In fact, many of these tools only support a fraction of these patterns and the best of them only support about 50%. Specifically the limited support for the discriminator, and its generalization, the N-out-of-M-join, the state-based patterns (only COSA), the synchronization of multiple instances (no tool fully supports this), cancellation (esp. of activities), and messaging, is worth noting. Also, observe that Staffware and Mobile are the only workflow management systems adopting a non-synchronizing strategy that support implicit termination.

[3] Note that the modeling language of Mobile is extensible. The results only indicate the standard functionality. All design patterns described in this paper can be added to Mobile.

pattern	product					
	Vis. WF	Forté	MQSeries	Staffware	Verve	Meteor
Synch Merge	-	+/-	+	-	-	-
Multi-Merge	-	+	-	-	+	+
Discriminator	-	+/-	-	-	+	+/-
N-out-of-M	-	+/-	-	-	-	+/-
Arb cycles	-	+	-	+	+	+
Impl termination	-	-	+	+	-	-
MI with runtime	+	+	+	-	+	-
MI w/out runtime	+	+	-	-	+	-
MI with synch	-	-	+/-	-	-	-
State-based	-	-	-	-	-	-
Cancel Activity	-	-	-	+	-	-
Cancel Case	-	+	-	-	+	-
M'Communication	+	-	-	-	+/-	+
M'Coordination	+	-	-	-	+/-	+
Bulk M	+	-	-	-	+/-	-

Table 1: The main results for Visual WorkFlo, Forté, MQSeries/Workflow, Staffware, Verve, and Meteor.

pattern	product					
	COSA	InConcert	Changeng.	I-Flow	SAP/R3	Mobile[3]
Synch Merge	-	+	+/-	-	-	-
Multi-Merge	-	-	-	-	-	-
Discriminator	-	-	+/-	-	+	+
N-out-of-M	-	-	+/-	-	+	+
Arb cycles	+	-	+	+	-	-
Impl termination	-	+	-	-	-	+
MI with runtime	-	-	-	+	-	-
MI w/out runtime	-	-	-	+	-	-
MI with synch	-	-	-	-	-	-
State-based	+	-	-	-	-	+/-
Cancel Activity	+	-	-	-	-	-
Cancel Case	-	-	+	+	-	-
M'Communication	+	-	-	-	+	-
M'Coordination	+	-	-	-	+	-
Bulk M	-	-	-	-	-	-

Table 2: The main results for COSA, InConcert, Changengine, I-Flow, SAP R/3, and Mobile.

4 Epilogue

This paper presented an overview of workflow patterns, emphasizing the control perspective, and discussed to what extent current commercially available workflow management systems could realize such patterns. Typically, when confronted with questions as to how certain complex patterns need to be implemented in their product, workflow vendors respond that the analyst may need to resort to the application level, the use of external events or database triggers. This however defeats the purpose of using workflow engines in the first place.

Through the discussion in this paper we hope that we not only have provided an insight into the shortcomings, comparative features and limitations of current workflow technology, but also that the patterns presented can provide a direction for future developments.

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

References

[Aal98a] W.M.P. van der Aalst. Chapter 10: Three Good reasons for Using a Petri-net-based Workflow Management System. In T. Wakayama et al., editor, *Information and Process Integration in Enterprises: Rethinking documents*, The Kluwer International Series in Engineering and Computer Science, pages 161–182. Kluwer Academic Publishers, Norwell, 1998.

[Aal98b] 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.

[AH00] W.M.P. van der Aalst and A.H.M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based Approach. *Information Systems*, 25(1):43–69, 2000.

[Cas98] R. Casonato. Gartner group research note 00057684, production-class workflow: A view of the market. http://www.gartner.com, 1998.

[CCPP95] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual Modeling of Workflows. In M.P. Papazoglou, editor, *Proceedings of the OOER'95, 14th International Object-Oriented and Entity-Relationship Modelling Conference*, volume 1021 of *Lecture Notes in Computer Science*, pages 341–354. Springer-Verlag, December 1995.

[CCPP98] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. *Data & Knowledge Engineering*, 24(3):211–238, January 1998.

[DE95] J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, United Kingdom, 1995.

[DKTS98] A. Doğaç, L. Kalinichenko, M. Tamer Özsu, and A. Sheth, editors. *Workflow Management Systems and Interoperability*, volume 164 of *NATO ASI Series F: Computer and Systems Sciences*. Springer, Berlin, Germany, 1998.

[EN93] 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.

[Fow97] M. Fowler. *Analysis Patterns: Reusable Object Models.* Addison-Wesley, Reading, Massachusetts, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[HK99] A.H.M. ter Hofstede and B. Kiepuszewski. Formal Analysis of Deadlock Behaviour in Workflows. Technical report, Queensland University of Technology/Mincom, Brisbane, Australia, April 1999. (submitted for publication).

[JB96] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation.* International Thomson Computer Press, 1996.

[KHB00] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On Structured Workflow Modelling. In B. Wangler and L. Bergman, editors, *Proceedings of the Twelfth International Conference on Advanced Information Systems Engineering (CAiSE'2000)*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445, Stockholm, Sweden, June 2000. Springer-Verlag.

[Kou95] T.M. Koulopoulos. *The Workflow Imperative.* Van Nostrand Reinhold, New York, 1995.

[Law97] P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition.* John Wiley and Sons, New York, 1997.

[LR99] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques.* Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.

[RZ96] D. Riehle and H. Züllighoven. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.

[Sch96] T. Schäl. *Workflow Management for Process Organisations*, volume 1096 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1996.

[SKM] 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.

[SL96] Software-Ley. *COSA User Manual.* Software-Ley GmbH, Pullheim, Germany, 1996.

[Sta97] Staffware. *Staffware 97 / GWD User Manual.* Staffware plc, Berkshire, United Kingdom, 1997.

[WFM96] WFMC. Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition, Brussels, 1996.

Contents

1 Introduction **2**

2 Workflow Patterns **4**

2.1 Basic Control Flow Patterns 5

2.2 Advanced Branching and Synchronization Patterns 8

2.3 Structural Patterns 13

2.4 Patterns involving Multiple Instances 16

2.5 Temporal Relations 21

2.6 State-based Patterns 22

2.7 Cancellation Patterns 28

2.8 Inter-Workflow Synchronization 30

3 Comparing Workflow Management Systems **35**

3.1 Introduction 35

3.2 Products 36

3.3 Results 40

4 Epilogue **43**