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] *School of Information Systems, 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: bkiepuszewski@yahoo.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, however their implementation is uncertain in 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, AH02, EN93, GHS95, JB96, Kou95, LR99, Law97, Sch96, WFM99, 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 variety of ways in which business processes are otherwise described. The absence of a universal organizational "theory", and standard business process modeling concepts, it is contended, explains and ultimately justifies the major differences in workflow languages - fostering up a "horses for courses" diversity in workflow languages. What is more, the comparison of different workflow products winds up being more of a dissemination of products and less of a critique of workflow language capabilities - "bigger picture" differences of workflow specifications are highlighted, as are technology, typically platform dependent, issues.

Workflow specifications can be understood, in a broad sense, from a number of different perspectives (see [AH02, 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, choice, parallelism and join synchronization. Activities in elementary form are atomic units of work, and in compound form modularize an execution order of a set of activities. The *data perspective* layers business and processing data on the control perspective. Business documents and other objects which flow between activities, and local variables of the workflow, qualify in effect pre- and post-conditions of activity execution. The *resource perspective* provides an organizational structure anchor to the workflow in the form of human and device roles responsible for executing activities. The *operational* perspective describes the elementary actions executed by activities, where the actions map into underlying applications. Typically, (references to) business and workflow data are passed into and out of applications through activity-to-application interfaces, allowing manipulation of the data within applications.

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

(AND and OR) and joins (AND and OR) - see [AH02, 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 the current capabilities of workflow languages nor insight into more complex requirements of business processes 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 a concrete form which keeps recurring in specific nonarbitrary 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.

To demonstrate solutions for the patterns, our recourse is a mapping to existing workflow language constructs. In some cases support from the workflow engine has been identified, and we briefly sketch implementation level strategies. Among the contemporary workflow management systems considered in this paper, none supports all the patterns. For those patterns that were supported, some had a straightforward mapping while others were demonstrable in a minority of tools.

It is important to note that the scope of our patterns is limited to static control flow, i.e., we do not consider patterns for resource allocation [KAV02], case handling [AB01], exception handling [CCPP98, KDB00], and transaction management [SAA99, GHS95].

Related work

Many languages have been proposed for the design and specification of workflow processes. Some of these languages are based on existing modeling techniques such as Petri nets and State charts. Other languages are system specific. Any attempt to give a complete overview of these languages and the patterns they support is destined to fail. Throughout this paper we will give pointers to concrete languages without striving for completeness. To our knowledge no other attempts have been made to collect a structured set of workflow patterns. This paper builds on [AHKB00a] where only four patterns are introduced. A previous version of this paper (evaluating only 12 systems but addressing more patterns) is available as a technical report [AHKB00b]. Moreover, the “Workflow Patterns Home Page” [AHKB] has been used to invite researcher, developers, and users to generate feedback. As a result, several authors have used our patterns to evaluate existing workflow management systems or newly designed workflow languages, e.g., in [Lav00] the OmniFlow environment is evaluated using 10 of our patterns, in [Hir01] our patterns are used to evaluate the CONDIS Workflow Management System, and in [VO01, VO02] the frequency of each of our patterns in real-life situations is investigated. Some of the patterns presented in this paper are related to the control-flow patterns described in [JB96]. However, the goal of [JB96] is to develop a workflow management systems that can be extended with new patterns rather than structuring and evaluating existing patterns. Our work is also related to investigations into the expressive power of workflow languages, cf. [BW99]. Other authors have coined the term workflow patterns but addressed different issues. In [WAH00] a set of workflow patterns inspired by Language/Action theory and specifically aiming at virtual communities is introduced. Patterns at the level of workflow architectures rather than control flow are given in [MB97].

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 *implementation* strategies. The problem component of a pattern describes why the construct is hard to realize in many of the workflow management systems available today. The implementation component, also referred to as solutions, 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).

Each workflow language can be formally described by a set of primitive modeling constructs, syntactical rules for composition, and the semantics of these constructs. In this paper, we will not present a new modeling language. Instead, we focus on workflow patterns that originate from business requirements. The semantics of these patterns are much less formal because we cannot assume a (formal) language. Moreover, the patterns are context-oriented, i.e., a workflow pattern typically describes certain business scenarios in a very specific context. The semantics of the pattern in this context is clear, while the semantics outside the context is undefined. Workflow patterns are typically realized in a specific language using one or more constructs available for this language. Please note the difference between the *semantics* of the pattern and the *realization* using a specific language. Sometimes workflow constructs available for a given language are not sufficient to realize a given pattern and workflow implementers have to resort to programming techniques such as event queuing, database triggers, etc to circumvent the limitations of a given workflow tool.

Patterns should be interpreted in a given context, i.e., assumptions about the environment which embeds the pattern are highly relevant. Consider for example the basic synchronization pattern (Pattern 3) often referred to as AND-join. It is a simple and well-understood pattern that describes a point in a workflow where multiple parallel subprocesses/activities converge into one single thread of control, thus synchronizing multiple threads. It is important to understand though that this pattern is clear and well-defined only in a very specific context, i.e. when we expect only one trigger from each of the incoming branches that we want to synchronize. This context is indeed the most common one, however not the only one possible and the simple synchronization pattern does not specify how synchronization should occur in a different context. Realizing the simple synchronization pattern is straightforward and involves usage of a language-specific synchronization construct, for example *synchronizer* in Verve, *rendezvous* in FileNet's Visual WorkFlo, *join* in MQSeries/Workflow, etc. However, each of these language-specific synchronization constructs behaves differently if this assumption about the context is dropped.

2.1 Basic Control Flow Patterns

In this section patterns capturing elementary aspects of process control are discussed. These patterns closely match the definitions of elementary control flow concepts provided by the WfMC in [WFM99]. 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*.

Implementation

- 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 typical implementation involves linking two activities with an unconditional control flow arrow.

□

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 threads 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.

Implementation

- All workflow engines known to us have constructs for the implementation of this pattern. 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. It is an assumption of this pattern that each incoming branch of a synchronizer is executed only once (if

this is not the case, then see Patterns 13-15 (Multiple Instances Requiring Synchronization)).
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.

Implementation

- All workflow engines available support constructs for the implementation of this pattern. Similarly to Pattern 2 one can identify two basic approaches: explicit AND-joins (e.g. Rendez-vous construct in Visual WorkFlo or Synchronizer in Verve) and implicit joins in an activity with more than one incoming transition (as in e.g. MQSeries/Workflow or Forté Conductor).

□

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

<u>Pattern</u> 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 is thoroughly evaluated by a senior employee.

Implementation

- Similarly to Pattern 2 (Parallel split) there are a number of basic strategies. Some workflow engines provide an explicit construct for the implementation of the exclusive choice pattern (e.g. Staffware, Visual WorkFlo). In some workflow engines (MQSeries/Workflow, Verve) the workflow designer has to emulate the exclusiveness of choice by specifying exclusive transition conditions. In another workflow product, Eastman, a post-processing rule list can be specified for an activity. After completion of the activity, the transition associated with the first rule in this list to evaluate to true is taken.

□

Pattern 5 (Simple Merge)
Description A point in the workflow process where two or more alternative branches come together without synchronization. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel (if this is not the case, then see Pattern 8 (Multi-merge) or Pattern 9 (Discriminator)).
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.

Implementation

- Given that we are assuming that parallel execution of alternative threads does not occur, 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 languages impose a certain level of structuredness to automatically guarantee that not more than one alternative thread is running at any point in time. Visual WorkFlo for example requires the merge construct to always be preceded by a corresponding exclusive choice construct (combined with some other requirements this then yields the desired behavior). In other languages workflow designers themselves are responsible for the design not having the possibility of parallel execution of alternative threads.

□

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 (Exclusive choice) 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 multi-choice.

Pattern 6 (Multi-choice)
Description A point in the workflow process where, based on a decision or workflow control data, a number of 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 multi-choice pattern can be implemented directly. However, there are workflow management systems which do not offer the possibility to specify conditions on transitions and which only offer pure AND-split and XOR-split building blocks (e.g. Staffware).
Implementation

- 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 only supply constructs to implement the parallel split and the exclusive choice, the implementation of the multi-choice has to be achieved through 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.

- A solution similar to the previous one is obtained by reversing the order of the parallel split pattern and the exclusive choice pattern. 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.

□

It should be noted that there is a trade-off between implementing the multi-choice as in Workflow A of Figure 1 or as in Workflow C of this figure. The solution depicted in Workflow A (assuming that the Workflow language allows for such an implementation) is much more compact and therefore more suitable for end-users. However, automatic verification of the workflow (i.e. checking for existence of deadlocks, etc.) is not possible for such solutions without additional knowledge of dependencies between the transition conditions (in Workflow C, the workflow designer has typically eliminated impossible combinations; this transformation is necessary, though not necessarily sufficient in itself, for enabling automatic verification).

Today's workflow products can handle the multi-choice pattern quite easily. Unfortunately, the implementation of the corresponding merge construct is much more difficult to realize. This merge construct, the subject of the next pattern, should have the capability to synchronize parallel flows and to merge alternative flows. The difficulty is to decide when to synchronize and when to merge. As an example, consider the simple workflow model shown in Figure 2. After

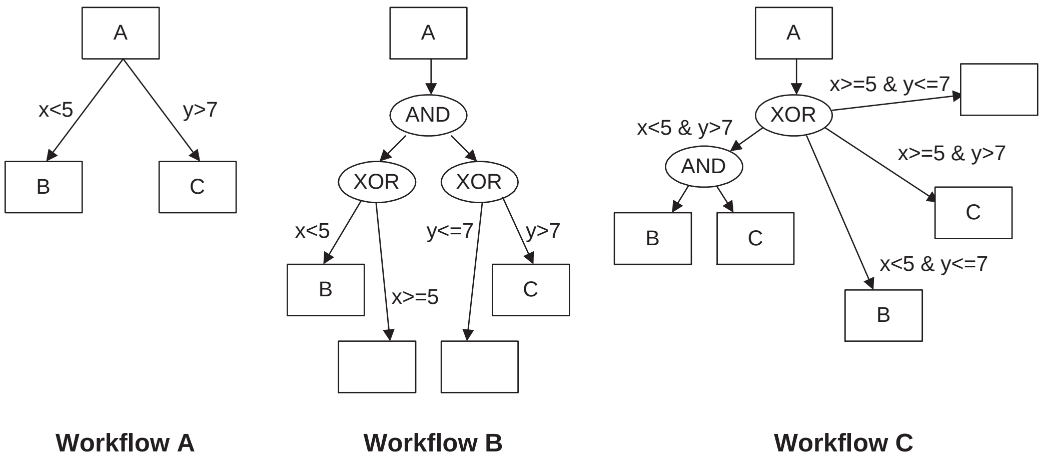

Figure 1: Design patterns for the multi-choice.

activity A finishes, either B or C, or *both* B and C, or *neither* B nor C will be executed. Hence, we would like to achieve the following traces: $ABCD$, $ACBD$, ABD, ACD, and A (these should be all the possible completed traces). The use of a simple synchronization construct leads to potential deadlock, while the use of a merge construct as provided by some workflow engines may lead to multiple execution of activity D (in case both B and C were executed).

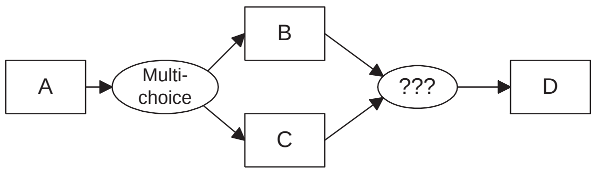

Figure 2: How do we want to merge here?

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. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete.
Synonyms Synchronizing join.
Examples

- Extending the example of Pattern 6 (Multi-choice), 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. Generally speaking, this type of merge needs to have some capacity to be able to determine whether it may (still) expect activation from some of its branches.
Implementation

- 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 Eastman, "non-parallel work items routed to Join worksteps bypass Join Processing" (p. 109 of [Sof98]), hence an XOR-split followed by an AND-join does not have to lead to deadlock. For example, if in the workflow of Figure 2 the multi-choice is replaced by an XOR-split and the merge construct by an AND-join, activity D would be reached. However, in Eastman, if an XOR-split is placed after activity B which leads to the AND-join but also to an empty (final) task, then activity D would not be reached if after executing B a choice is made for this empty task and the workflow would be in deadlock. Hence, joins require some information about how many active threads to expect under certain circumstances.

- In other workflow engines the implementation of the synchronizing merge typically is not straightforward. The only solution 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 (Multi-choice)). This way we can easily synchronize corresponding branches by using AND-join and standard merge constructs.

□

The next two patterns can be applied in contexts where the assumption made in Pattern 5 (Simple merge) does not hold, i.e. they can deal with merge situations where multiple incoming branches may run in parallel. As an example, consider the simple workflow model depicted in Figure 3. If a standard synchronization construct (Pattern 3 (Synchronization)) is used as a merge construct, activity D will be started once, only after activities B *and* C are completed. Then, all possible completed traces of this workflow are $ABCD$ and $ACBD$. There are situations though where it is desirable that activity D is executed once, but started after *either* activity B or activity C is completed (as to avoid waiting unnecessarily for the other activity to finish). All possible completed traces would then be $ABCD$, $ACBD$, $ABDC$, $ACDB$. Such a pattern will be referred to as a *discriminator*. Another scenario which may occur is one where activity D is to be executed *twice*, after activity B is completed and also after activity C is completed. All possible completed traces of this workflow will be $ABCDD$, $ACBDD$, $ABDCD$, and $ACDBD$. Such a pattern will be referred to as a *multi-merge*.

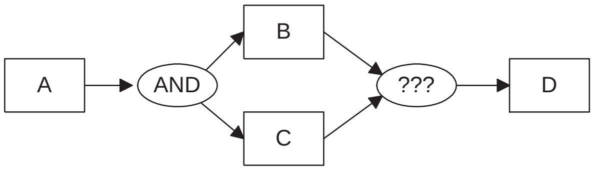

Figure 3: How do we want to merge here?

<u>Pattern</u> 8 (Multi-merge)
Description 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 *for every activation of every incoming branch.*
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 The use of a standard merge construct as provided by some workflow products to implement this pattern often leads to undesirable results. Some workflow products (e.g. Staffware, I-Flow) will not generate a second instance of an activity if another instance is still running, while e.g. HP Changengine will never start a second instance of an activity. Finally, in some workflow products (e.g. Visual WorkFlo, SAP R/3 Workflow) it is not even possible to use a merge construct in conjunction with a parallel split as in the workflow of Figure 3 due to syntactical restrictions that are imposed.
Implementation

- The merge constructs of Eastman, Verve Workflow and Forté Conductor can be used directly to implement this pattern.

- 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 4 for a simple example). If the multi-merge is part of a 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 Pattern 14 (Multiple Instances with a Priori Runtime Knowledge) and Pattern 15 (Multiple Instances Without a Priori Runtime Knowledge).

- An interesting solution is offered by the case-handling system FLOWer. FLOWer allows for dynamic subplans. A dynamic subplan is a subprocess with a variable number of instances. Moreover, the number of instances can be controlled dynamically through a variable. This way it is possible to indirectly model the multi-merge.

□

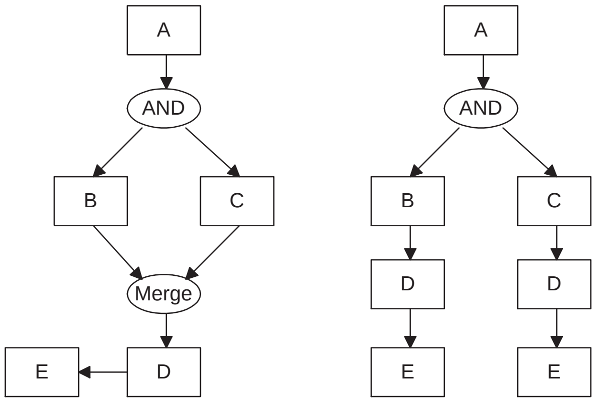

Figure 4: Typical implementation of multi-merge pattern.

<u>Pattern</u> **9 (Discriminator)**
Description The discriminator is a point in a workflow process that waits for one of the 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 (which is important otherwise it could not really be used in the context of a loop).
Examples

- 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 Most workflow engines do not have a construct that can be used for a direct implementation of the discriminator pattern. As mentioned in Pattern 8 (Multi-merge), the standard merge construct in some workflow engines (e.g. Staffware, I-Flow) will not generate the second instance of an activity if the first instance is still active. This does *not* provide a solution for the discriminator, however, since if the first instance of the activity finishes before an attempt is made to start it again, a second instance *will* be created (in terms of Figure 3 this would mean that e.g. a trace like $ABDCD$ is possible).

Implementation

- A regular join construct in Changengine has a semantics similar to that of the discriminator.

- There is a special construct that implements the discriminator semantics in Verve (in fact we adopted this term from this product). This construct has many incoming branches and one outgoing branch. When one of the incoming branches finishes, the subsequent activity is triggered and the discriminator changes its state from "ready" to "waiting". From then on it waits for all remaining incoming branches to complete. When that has happened, it changes its state back to "ready". This construct provides a direct implementation option for the discriminator pattern, however, it does not work properly when used in the context of a loop (once waiting for the incoming branches to complete, it ignores additional triggers from the branch that fired it).

- In SAP R/3 Workflow (version 4.6C) for forks (a combination of an AND-split and an AND-join) it is possible to specify the number of branches that have to be completed for the fork to be considered completed. Setting this number to one realizes a discriminator except that 1) the branches that have not been completed receive the status "logically deleted" and 2) the fork restricts the form that parallelism/synchronization can take.

- The discriminator semantics can be implemented in products supporting *Custom Triggers*. For example in Forté Conductor a custom trigger can be defined for an activity that has more than one incoming transition. Custom triggers define the condition, typically using some internal script language, which when satisfied should lead to execution of a certain activity. Such a script can be used to achieve a semantics close to that of a discriminator (again, in the context of a loop such a script may be more complicated). The downside of this approach is that the semantics of a join that uses custom triggers is impossible to determine without carefully examining the underlying trigger scripts. As such, the use of custom triggers may result in models that are less suitable and hard to understand.

- Some workflow management systems (e.g., FLOWer) allow for data-dependent executions. In FLOWer it is possible to have a so-called milestone which waits for a variable to be set. The moment the variable is set, processing of the parallel thread containing the milestone will continue. The reset functionality is realized through using so-called sequential/dynamic subplans rather than iteration.

- Typically, in other workflow engines the discriminator is impossible to implement directly in the workflow modeling language supplied.

□

To realize a discriminator that behaves properly in loops is quite complicated and this may be the reason why it has not been implemented in its most general form in any of the workflow products referred to in this paper. The discriminator needs to keep track of which branches have completed (and how often in case of multiple activations of the same branch) and resets itself when it has seen the completion of each of its branches.

Note that the discriminator pattern can easily be generalized for the situation when an activity should be triggered only after n out of m incoming branches have been completed. Similarly to the basic discriminator all remaining branches should be ignored. In the literature, this type of discriminator has been referred to as a partial join (cf. [CCPP95]). Implementation approaches to this pattern are similar to those for the basic discriminator when *custom triggers* can be used and SAP R/3 Workflow's approach already allowed the number of branches that need to be completed to be more than one. In languages that provide direct support for the basic discriminator (e.g. Verve Workflow) an n-out-of-m join can be realized with the additional use of a combination of AND-joins and AND-splits (the resulting workflow definition becomes large and complex though). An example of the realization of a 2-out-of-3 join is shown in Figure 5.

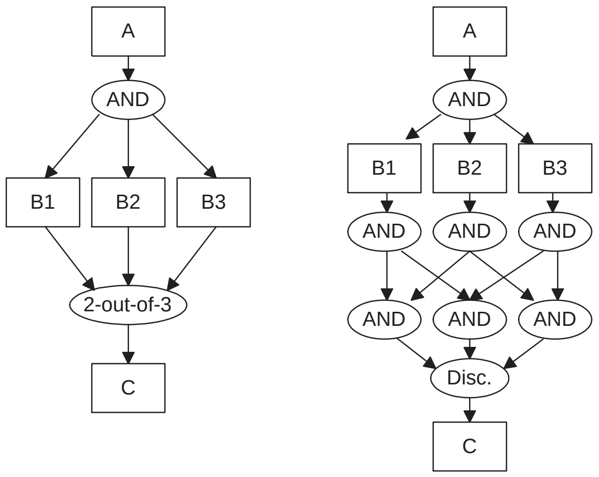

Figure 5: Implementation of a 2-out-of-3-join using the basic discriminator.

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.

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 10 (Arbitrary Cycles)
Description A point in a workflow process where one or more activities can be done repeatedly.
Synonyms Loop, iteration, cycle.
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,FLOWer) or through a special loop construct (Visual WorkFlo, SAP R/3 Workflow).
Implementation

- 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 (Multiple Instances With a Priori Runtime Knowledge)). The conversion is done through auxiliary variables and/or node repetition. An analysis of such conversions and an identification of some situations where they cannot be done can be found in [KHB00]. Figure 6 provides an example of an arbitrary workflow converted to a structured workflow. Such a structured workflow can be implemented directly in workflow engines such as MQSeries/Workflow or Visual WorkFlo that do not have direct support for arbitrary cycles. 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.

□

Remark 2.1
The rightmost workflow in Figure 6 requires a slight adaptation, shown in Figure 7, in

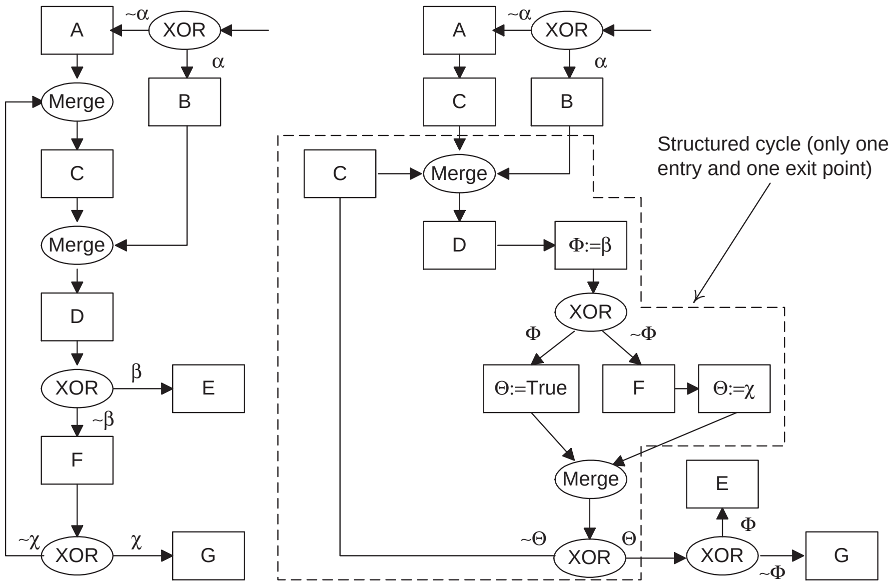

Figure 6: Example of implementation of arbitrary cycles.

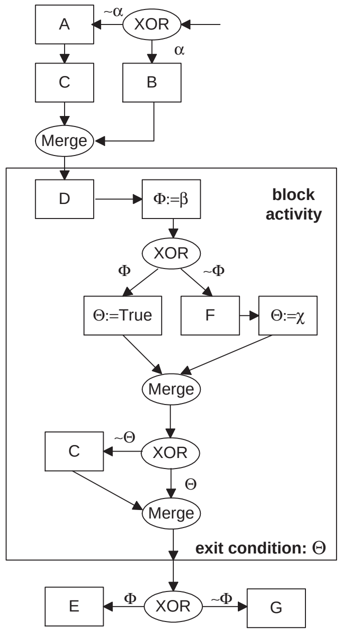

Figure 7: Transformation of structured cycle of Figure 6 to block structure in MQSeries/Workflow.

order to be realizable in block structured languages such as MQSeries Workflow. This adaptation is required as Θ is the post-condition for this block. If Θ evaluates to false, the block should start again, and the next activity should be the initial activity of the block (but the activity named C which follows the evaluation of Θ is not initial). The transformation though is straightforward and always possible for structured cycles.

□

Note that some authors (e.g., [AM00]) claim that any loop is an exception and should not be modeled explicitly. As an alternative, typically a facility to jump back to any previous state in the workflow (cf. the backward linear jumps in [AM00]) is proposed. This approach has been implemented in FLOWer [AB01]. In FLOWer any activity has a so-called "redo role". The redo role specifies who is allowed to undo the activity and jump backwards in the process. This allows for the implicit modeling of loops.

Another example of a requirement imposed by some workflow products 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 of these ending nodes is completed. Again, many 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 11 (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).
Problem Most workflow engines terminate the process when an explicit *Final* node is reached. Any current activities that happen to be running at that time will be aborted.
Implementation

- Some workflow engines (Eastman, Staffware, MQSeries/Workflow, InConcert) support this pattern directly as they would terminate a (sub)process when there is nothing else to be done.

- For workflow products that do not support this pattern directly, 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 is 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 to a model with explicit termination. A detailed analysis of which workflow model can be converted to an equivalent model that has only one terminating node is beyond the scope of this paper.

□

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 multiple threads of execution referring to a shared definition. From a practical point of view it means that an activity in a workflow graph can have more than one running, active instance at the same time. As we will see, such behavior may be required in certain situations. The fundamental problem with the implementation of these patterns 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.

When considering multiple instances there are two types of requirements. The first requirements has to do with the ability to launch multiple instances of an activity or a subprocess. The second requirement has to do with the ability to synchronize these instances and continue after all instances have been handled. Each of the patterns needs to satisfy the first requirement. However, the second requirement may be dropped by assuming that no synchronization of the instances launched is needed. This assumption is somewhat related to patterns 8 (Multi-merge) and 11 (Implicit Termination). The Multi-merge also allows for the creation of multiple instances without any synchronization facilities. If instances that are created are not synchronized, then termination of each of these instances is implicit and not coordinated with the main workflow.

If the instances need to be synchronized, the number of instances is highly relevant. If this number is fixed and known at design time, then synchronization is rather straightforward. If however, the number of instances is determined at run-time or may even change while handling the instances, synchronization becomes very difficult. Therefore, we identify three patterns with synchronization. If no synchronization is needed, the number of instances is less relevant: Any facility to create instances within the context of a case will do. Therefore, we only present one pattern for multiple instances without synchronization.

Pattern 12 (Multiple Instances Without Synchronization)
Description Within the context of a single case (i.e., workflow instance) multiple instances of an activity can be created, i.e., there is a facility to spawn off new threads of control. Each of these threads of control is independent of other threads. Moreover, there is no need to synchronize these threads.
Synonyms Multi threading without synchronization, Spawn off facility
Examples

- A customer ordering a book from an electronic bookstore such as Amazon may order multiple books at the same time. Many of the activities (e.g., billing, updating customer records, etc.) occur at the level of the order. However, within the order multiple instances need to be created to handle the activities related to one individual book (e.g., update stock levels, shipment, etc.). If the activities at the book level do not need to be synchronized, this pattern can be used.

Implementation

- 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 the use of parallel splits without corresponding joins and allows triggering of activities that are already active. This is possible in languages such as Forté and Verve. This solution is illustrated by Workflow A in Figure 8.
- 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*. COSA has a similar facility, one workflow may contain multiple concurrent flows that are created through an API and share information.
- In most workflow management systems the possibility exists to create new instances of a workflow process through some API. This allows for the creation of new instances by calling the proper method from activities inside the main flow. Note that this mechanism works. However, the system maintains no relation between the main flow and the instances that are spawned off.

□

Pattern 12 is supported by most workflow management systems. The problem is not to *generate* multiple instances, the problem is to *coordinate* them. As explained before, it is not trivial to synchronize these instances. Therefore, we will present three patterns involving the synchronization of concurrent threads.

The simplest case is when we know, during the design of the process, the number of instances that will be active during process execution. In fact, this situation can be considered to be a combination of patterns 2 (Parallel Split) and 3 (Synchronization) were all concurrent activities share a common definition.

<u>Pattern</u> 13 (Multiple Instances With a Priori Design Time Knowledge)
Description For one process instance an activity is enabled multiple times. The number of instances of a given activity for a given process instance is known at design time. Once all instances are completed some other activity needs to be started.
Examples

- The requisition of hazardous material requires three different authorizations.

Implementation

- If the number of instances is known a priori during design time, then a very simple implementation option is to replicate the activity in the workflow model preceding it with a construct used for the implementation of the parallel split pattern. Once all activities are completed, it is simple to synchronize them using a standard synchronizing construct.

□

It is simple enough to model multiple instances when their number is known a priori, as one simply replicates the task in the process model. However, if this information is not known, and the number of instances cannot be determined until the process is running, this technique cannot be used. The next two patterns consider the situation when the number of instances is not known at design time. The first pattern considers the situation where it is possible to determine the number of instances to be started *before* any of these instances is started.

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 varies 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. Once all instances are completed some other activity needs to be started.
Examples

- In the review 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. Only if all reviews have been returned, processing is continued.

- For the processing of an order for multiple books, the activity *check_availability* is executed for each individual book. The shipping process starts if the availability of each book has been checked.

- 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.

- When authorizing a requisition with multiple items, each item has to be authorized individually by different workflow users. Processing continues if all items have been handled.

Problem As the number of instances of a given activity is not known during the design we cannot simply replicate this activity in a workflow model during the design stage. Currently only a few workflow management systems allow for multiple instances of a single activity at a given time, or offer a special construct for the multiple activation of one activity for a given process instance, such that these instances are synchronized.
Implementation

- If the workflow engine supports multiple instances directly (cf. Forté and Verve), we can try and use the solution illustrated in Workflow A in Figure 8. However, activity E in this model will possibly be started before all instances of activity B are completed. To achieve proper synchronization one needs to resort to techniques well beyond the modeling power of these languages. For example, it may be possible to implement activity B such that once it is completed, it sends an event to some external event queue. Activity E can be preceded by another activity that consumes the events from the queue and triggers E only if the number of events in the queue is equal to the number of instances of activity B (as pre-determined by activity A). This solution is very complex, may have some concurrency problems, and for the end-user it is totally unclear what the true semantics of the process is.

- Similar problems occur when using the *Release* construct of Visual WorkFlo, the *Chained Process Node* of I-Flow, the multiple subflows of COSA, or some API to invoke the subprocess as part of an activity in a process. In each of these systems, it is very difficult to synchronize concurrent subprocesses.

- Some workflow engines offer a special construct that can be used to instantiate a given number of instances of an activity. An example of such a construct is the *Bundle* concept that was available in FlowMark, version 2.3 (it is not available in MQSeries/Workflow version 3.3). 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. Once all instances in a bundle are completed, the next activity is started. The bundle construct provides a very clear and straightforward solution to the problem (see Workflow D in Figure 8). A similar concept is provided in SAP R/3 Workflow through "Table-driven Dynamic Parallel Processing".

- If there is a maximum number of possible instances, then a combination of AND-splits and XOR-splits can be used to obtain the desired routing. An 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 an AND-split with the corresponding cardinality. The drawback of this solution is that the resulting workflow model can become large and complex and the maximum number of possible instances needs to be known in advance (see Workflow C in Figure 8).

- As in many cases, the desired routing behavior can be supported quite easily by making it more sequential. Simply use iteration (cf. Pattern 10 (Arbitrary Cycles)) to activate instances of the activity *sequentially.* Suppose that activity A is followed by n instantiations of B followed by E. First execute A, then execute the first instantiation of B. Each instantiation of B is followed by an XOR-split to determine whether another instantiation of B is needed or that E is the next step to be executed. This solution is fairly straightforward. However, the n instantiations of B are not executed in parallel but in a fixed order (see workflow B in Figure 8). In many situations this is not acceptable. Recall the example of the refereeing process of papers. Clearly, it is not acceptable that the second referee has to wait until the first referee completes his/her review, etc.

□

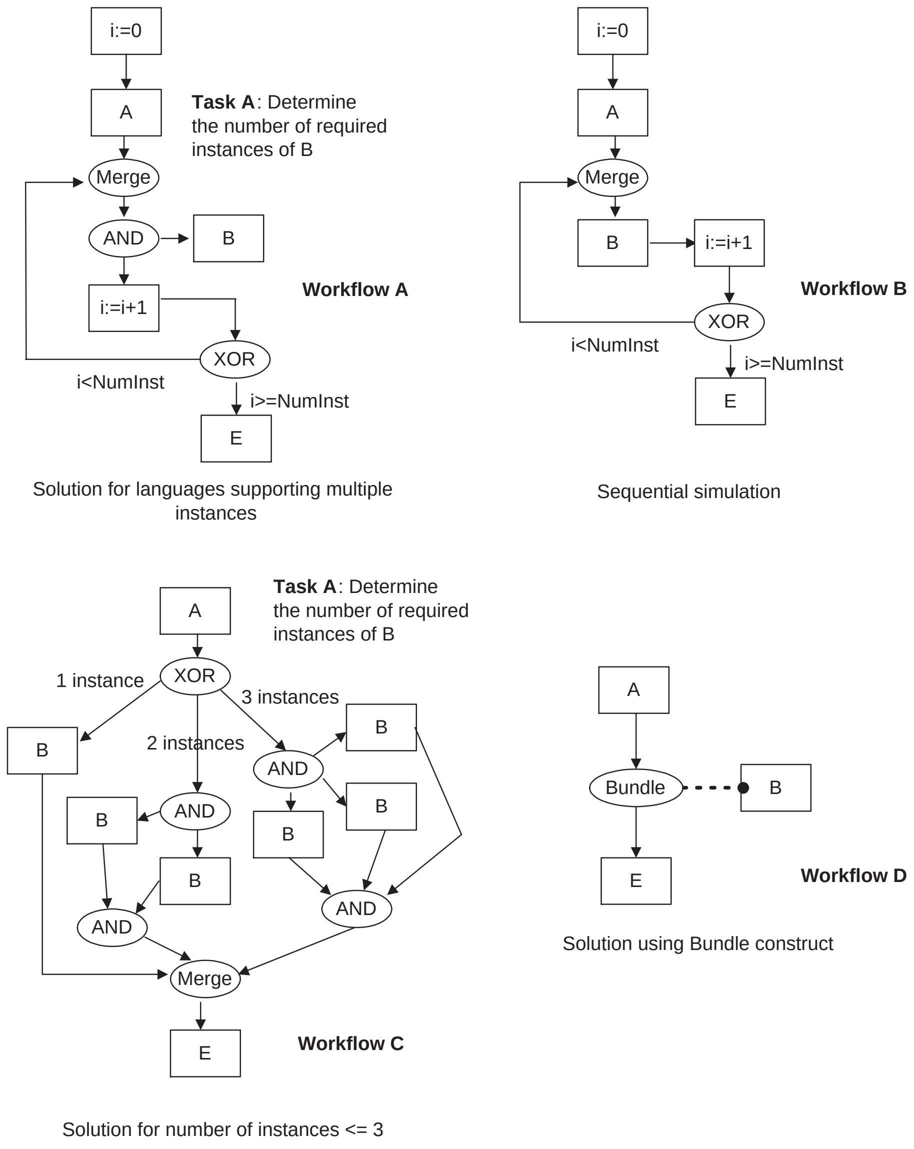
i:=0
A
Task A: Determine the number of required instances of B
Merge
AND
B
i:=i+1
Workflow A
XOR
i<NumInst
i>=NumInst
E
Solution for languages supporting multiple instances
i:=0
A
Merge
B
i:=i+1
Workflow B
XOR
i<NumInst
i>=NumInst
E
Sequential simulation
A
Task A: Determine the number of required instances of B
XOR
1 instance
3 instances
2 instances
B
AND
Merge
E
Workflow C
Solution for number of instances <= 3
A
Bundle
B
E
Workflow D
Solution using Bundle construct

Figure 8: Design patterns for multiple instances.

Finally, we would like to present a pattern which is typically the hardest to implement. In it the number of instances in a process is determined in a totally dynamic manner rendering solutions such as e.g. the use of the *Bundle* concept inappropriate.

<u>Pattern</u> **15 (Multiple Instances Without 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 is it known at any stage during runtime, before the instances of that activity have to be created. Once all instances are completed some other activity needs to be started. The difference with Pattern 14 is that even while some of the instances are being executed or already completed, new ones can 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. After each delivery, 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. After processing all deliveries, the requisition has to be closed.

- For the processing of an insurance claim, zero or more eyewitness reports should be handled. The number of eyewitness reports may vary. Even when processing eyewitness reports for a given insurance claim, new eyewitnesses may surface and the number of instances may change.

Problem Some workflow engines provide support for generating multiple instances only if the number of instances is known at some stage of the process. This can be compared to a "for" loop in procedural languages. However, these constructs are of no help to processes requiring "while" loop functionality.
Implementation

- FLOWer is one of the few systems directly supporting this pattern. In FLOWer it is possible to have dynamic subplans. The number of instances of each subplan can be changed at any time (unless specified otherwise).

- This pattern is a generalization of Pattern 14 (Multiple Instances With a Priori Runtime Knowledge). Some implementation strategies are also applicable here. Specifically, the creation part of this pattern may easily be implemented if the engine supports multiple instances directly. Similarly we may also provide an implementation using special constructs for "spawning off" new processes or using APIs to do that. However, as with Pattern 14, synchronization of the instances is very hard to achieve, in fact in this pattern it is even harder as there is no count of spawned-off activities readily available. Consider for example Workflow A in Figure 9. Since *NumInst* may vary while instances of B are executed, the implementation of this construct is more involved. Dynamically, the number of instances (to be) activated needs to be compared with the number of instances completed. This can be implemented by a precondition and an event queue connected to E which counts the number of completed instances of B, i.e., each instance of B generates an event for E when it completes. Activity E has a precondition comparing the number of instances launched and the number of instances completed.

- If the language supports multiple instances *and* supports a decomposition concept with implicit termination (hence a decomposition is only considered to be finished when all its 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 (see Workflow B in Figure 9). Here, activity B will be invoked many times, and 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. We find this approach to be a very natural solution to the problem, however, none of the languages included in our review supports both multiple instances and a decomposition concept with implicit termination.

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

□

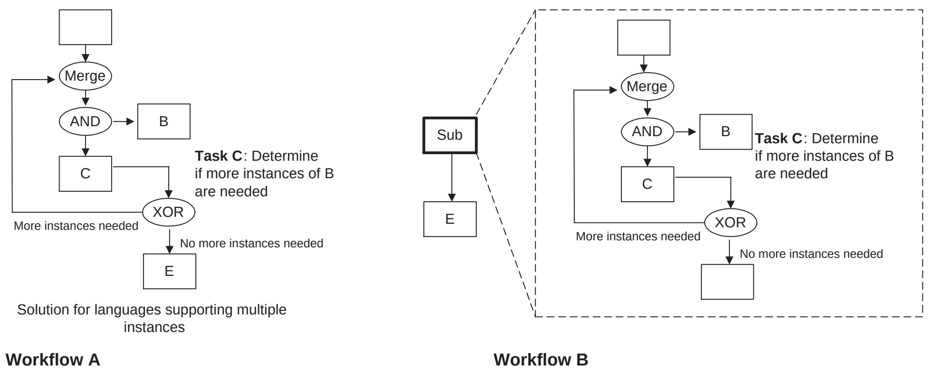
Merge
AND
B
C
Task C: Determine if more instances of B are needed
XOR
More instances needed
No more instances needed
E
Solution for languages supporting multiple instances
Workflow A
Sub
E
Merge
AND
B
Task C: Determine if more instances of B are needed
C
XOR
More instances needed
No more instances needed
Workflow B

Figure 9: Design patterns for multiple instances.

As a side note to multiple instances patterns we would like to point out an interesting problem associated with the use of Pattern 3 (Synchronization) and Pattern 8 (Multi-merge). Consider the simple workflow shown in Figure 10. The *multi-merge* construct will cause both activities A and B to be instantiated twice. We would then like activity C to be instantiated twice as well. Different engines, however, behave differently when a synchronization construct is used in a process where multiple triggering of activities is allowed. For example, workflow products such as Verve and HP Changengine support a synchronizer notion whereby multiple triggering by the same activity is ignored. So in this example the 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. As a result, activity C may be performed once or twice depending on the sequence of completion of instances of A and B. In Staffware the problem is even more complicated by the fact that Staffware's AND-join is not commutative. As input of an AND-join, there is a transition (graphically represented by a solid line) that waits till other transitions are released and there are transitions (graphically represented by dashed lines) that represent threads that have to be waited upon. Multiple signals from the former type of transition are ignored, while multiple signals from the latter type of transition are remembered.

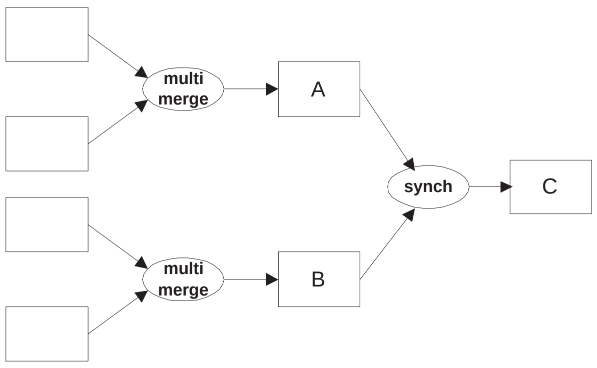

Figure 10: Workflow depicting a basic synchronization problem.

2.5 State-based Patterns

In real workflows, most workflow instances are in a state awaiting processing rather than being processed. Many 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 will use the well-known Petri-net notation [RR98, 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 [SL99] and Income [Pro98]).

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 11 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 (the envelope notation denotes that activity C requires an external trigger) 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.

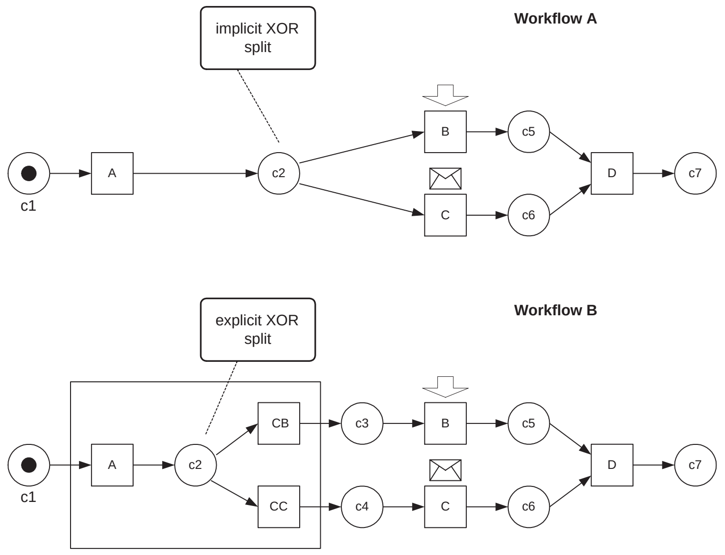

Figure 11: Illustrating the difference between implicit (Workflow A) and explicit (Workflow B) XOR-splits.

For the readers familiar with Petri nets, the difference between both workflows becomes clear when considering the places in Figure 11. In Workflow A each case in-between A and B or C is in the state represented by place $c2$. In this state both B and C are enabled. In Workflow B each case in-between A and B or C is either in the state represented by place $c3$ or in the state represented by place $c4$. If place $c3$ is marked, only B is enabled. If place $c4$ is marked, only C is enabled. The only way to distinguish both situations is by explicitly considering the states in between subsequent activities. The modeling languages used by contemporary workflow management systems typically abstract from states between subsequent activities, and hence have difficulties modeling implicit choices.

Pattern 16 (Deferred Choice)
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 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, deferred XOR-split.
Examples

- At certain points during the processing of insurance claims, quality assurance audits are undertaken at random by a unit external to those processing the claim. The occurrence of an audit depends on the availability of resources to undertake the audit, and not on any knowledge related to the insurance claim. Deferred Choices can be used at points where an audit might be undertaken. The choice is then between the audit and the next activity in the processing chain. The audit activity triggers the next activity to preserve the processing chain.

- Consider activity A in Figure 11 to represent the activity *send_questionnaire*, and activities B and C, the activities *time_out* and *process_questionnaire* respectively. 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.

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

- Business trips require approval before being booked. There are two ways to approve a task. Either the department head approves the trip (activity A_1) or both the project manager (activity A_{21}) and the financial manager (activity A_{22}) approve the trip. The latter two activities are executed sequentially and the choice between A_1 on the one hand and A_{21} and A_{22} on the other hand is implicit, i.e., at the same time both activity A_1 and activity A_{21} are offered to the department head and project manager respectively. The moment one of these activities is selected, the other one disappears.

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

- COSA is one of the few systems that directly supports the deferred choice. Since COSA is based on Petri nets it is possible to model implicit choices as indicated in Figure 11(A). Some systems offer partial support for this pattern by offering special constructs for a deferred choice between a user action and a time out (e.g., Staffware) or two user actions (e.g., FLOWer).

- Assume that the workflow language being used supports cancellation of activities through either a special transition (for example Staffware, see Pattern 19 (Cancel Activity)) or through an API (most other engines). Cancellation of an activity means that the activity is being removed from the designated worklist as long as it has not been started yet. The deferred choice 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 deferred choice between B and C in Figure 11 (Workflow 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. Workflow A of Figure 12 shows the corresponding workflow model. Note that the solution does not always work because B and C can be selected/executed concurrently.

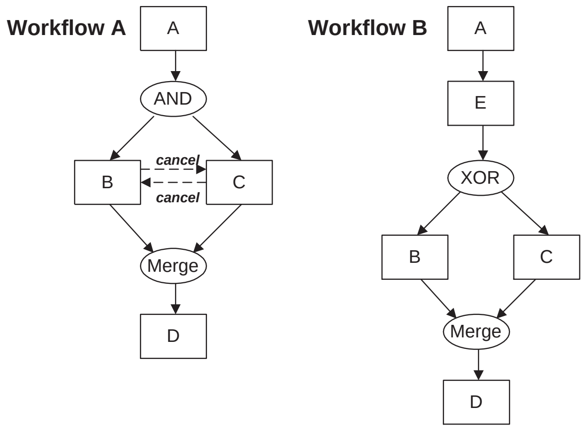

Figure 12: Strategies for implementation of deferred choice.

- Another solution to the problem is to replace the deferred choice 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 11. 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. Workflow B of Figure 12 shows the corresponding workflow model. Note that the solution moves part of the routing to the application or task level. Moreover, this solutions assumes that the choice is made based on the type of trigger.

□

Typically, Patterns 2 (Parallel Split) and 3 (Synchronization) are 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 17 (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 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.
Implementation

- 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 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 combination of implementation constructs for the sequence and the exclusive choice patterns 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 and it is not clear how the choice is made. Moreover, the workflow model may become quite complex and large by enumerating all possible sequences. Workflow B in Figure 13 illustrates this solution in a case with three activities.

- By using implementation strategies for the deferred choice pattern (instead of an explicit XOR-split) the order does not need to be fixed before the execution starts, i.e. the implicit XOR-split allows for on-the-fly selection of the order. Unfortunately, the resulting model typically has a "spaghetti-like" structure. This solution is illustrated by Workflow C of Figure 13.

- 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 14 for an example where this construct is applied. Note that, unlike the other solutions, the structure of the model is not compromised.

□

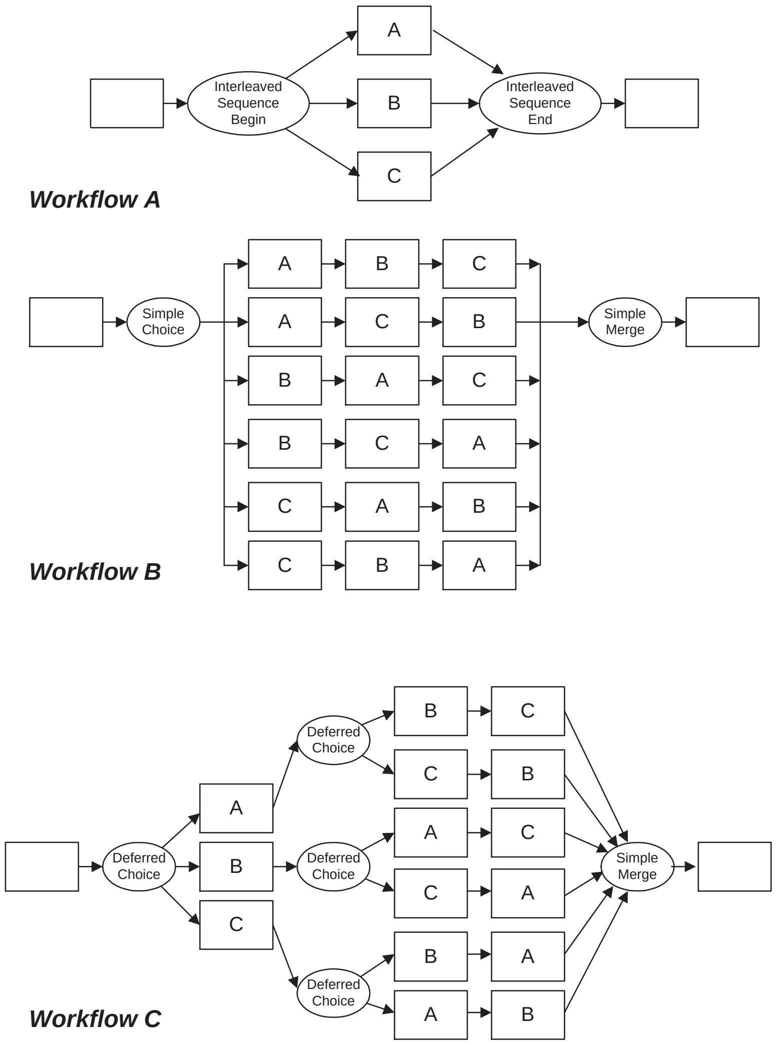

Figure 13: The implementation options for interleaving execution of A, B and C.

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 14 is only possible because mutual exclusion can be enforced by place *mutex* (i.e. state information shared among the activities). Pattern 16 (Deferred choice) is another

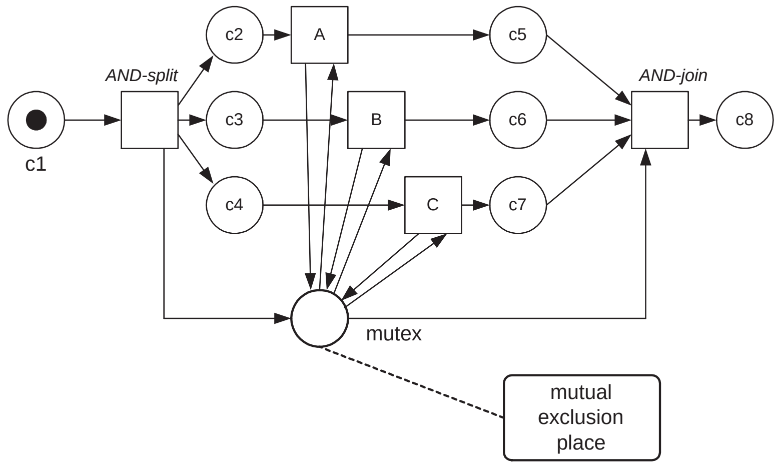

Figure 14: The execution of A, B, and C is interleaved by adding a mutual-exclusion place.

example of a construct which is hard to handle if one abstracts from the states in-between activities. The next pattern, Pattern 18 (Milestone), 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 15). 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*). Note that there is a deferred choice between *process_questionnaire* and *time_out* (Pattern 16). Based on the result of the evaluation (activity *evaluate*), the complaint is processed or not. Transitions *skip* and *processing_needed* have been added to model the explicit choice. Note that the choice between *skip* and *processing_needed* is not offered to the environment. This choice is made by the workflow management system, e.g., based on an attribute set in activity *evaluate*. 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*. Again two transitions (*OK* and *NOK*) have been added to model the explicit choice. Finally, activity *archive* is executed.

□

The construct involving activity *process_complaint* which is only enabled if place *c5* contains a token is called a milestone.

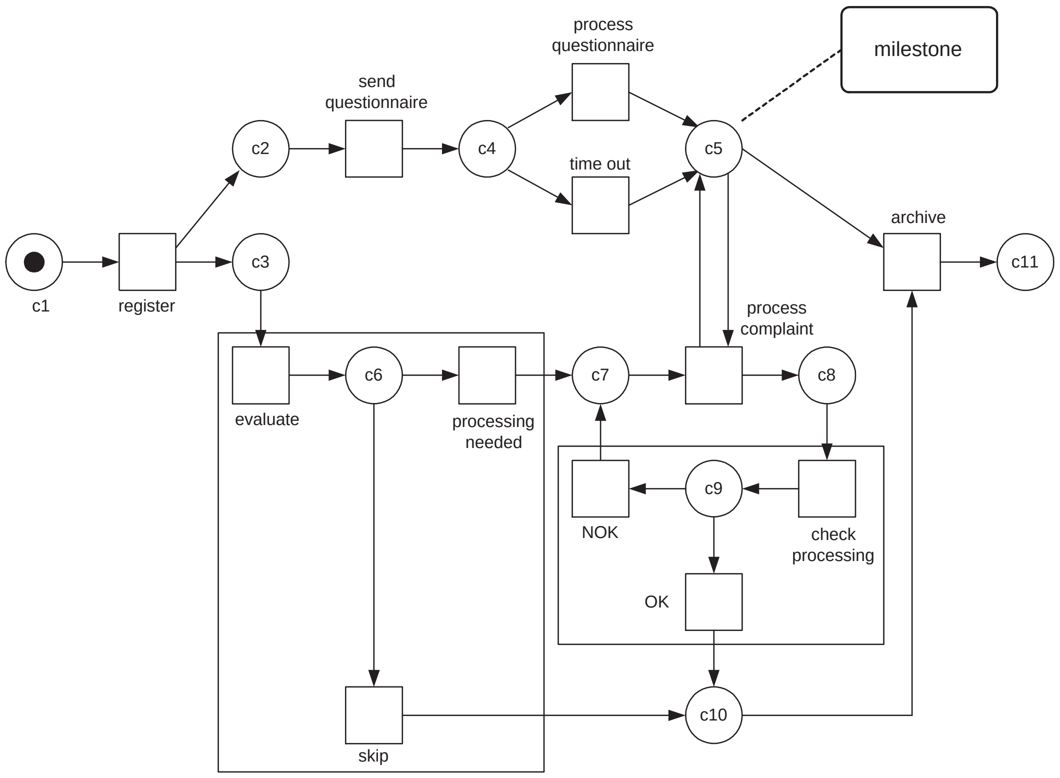

Figure 15: The state in-between the processing/time-out of the questionnaire and archiving the complaint (i.e. place *c5*) is an example of a milestone.

<u>Pattern</u> 18 (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 named 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 of B and A is not enabled after the execution of C. Figure 16 illustrates the pattern. The state in between B and C is modeled by place m. This place is a milestone for A. Note that A does not remove the token from M: It only tests the presence of a token.
Synonyms Test arc, deadline (cf. [JB96]), state condition, withdraw message.
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.

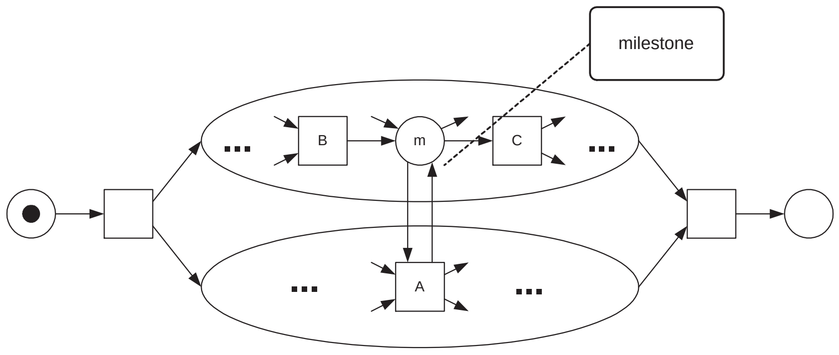

Figure 16: Schematical representation of a milestone.

- The construct involving activity *process_complaint* and place *c5* shown in Figure 15.

Problem The problem is similar to the problem mentioned in Pattern 16 (Deferred Choice): There is a race between a number of activities and the execution of some activities may disable others. In most workflow systems (notable exceptions are those based on Petri nets) once an activity becomes enabled, there is no other-than-programmatic way to disable it. A milestone can be used to test whether some part of the process is in a given state. Simple message passing mechanisms will not be able to support this because the disabling of a milestone corresponds to withdrawing a message. This type of functionality is typically not offered by existing workflow management systems. Note that in Figure 15 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. It is not possible to model such a construct by an AND-split/AND-join type of synchronization between the two parallel branches, because it is not known how many times a synchronization is needed.
Implementation

- 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, cf. Workflow A in Figure 17. Such a milestone can be realized using Pattern 16 (Deferred Choice). After executing B there is an implicit XOR-split with two possible subsequent activities: A and C. If A is executed, then the same implicit XOR-split is activated again. If C is executed, A is disabled by the implicit XOR-split construct. This solution is illustrated by Workflow B in Figure 17. Note that this solution only works if the execution of A is not restricted by other parallel threads. For example, the construct cannot be used to deal with the situation modeled in Figure 15 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. Clearly, a choice restricted by multiple parallel threads cannot be handled using Pattern 16 (Deferred Choice).

- 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. This solution is illustrated by Workflow C in Figure 17. Note that this way a "busy wait" is introduced and after enabling A it cannot be blocked anymore, i.e., the execution of C does not influence running or enabled instances of A. Using Pattern 19 (Cancel Activity), A can be withdrawn once C is started. 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.

□

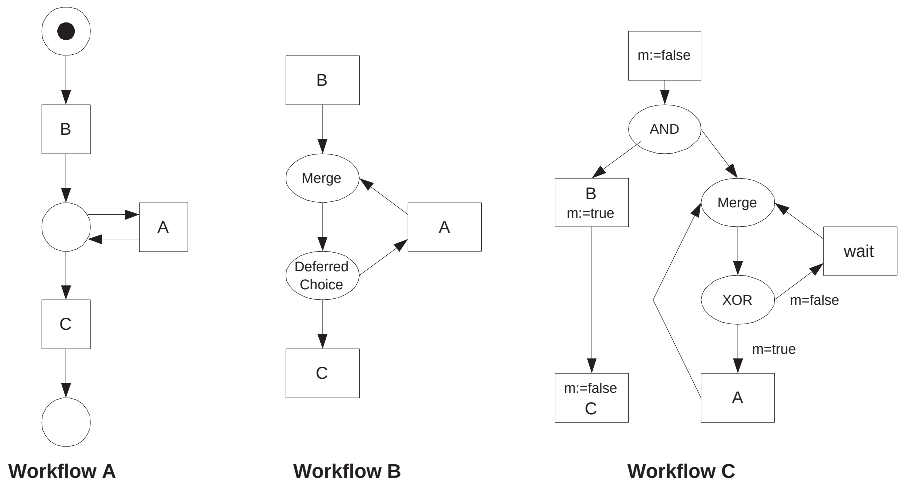

Figure 17: The Milestone pattern in its simplest form: implemented using a Petri net (Workflow A), implemented using a deferred choice (Workflow B), and implemented using a busy wait (Workflow C).

It is interesting to think about the reason why many workflow products have problems dealing with the state-based patterns. Systems that abstract from states are typically based on messaging, i.e. if an activity finishes, it notifies or triggers other activities. This means that activities are *enabled* by the receipt of one or more messages. The state-based patterns 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 the state-based patterns further complicates the use of asynchronous communication mechanisms such as message passing using "negative messages" (e.g. messages to cancel previous messages).

2.6 Cancellation Patterns

The first solution described in Pattern 16 (Deferred Choice) 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. (See Figure 12: The dashed arrows correspond to withdrawals.) The following pattern describes this construct.

Pattern 19 (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 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 for 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.
Implementation

- If the workflow language supports Pattern 16 (Deferred Choice), then it is possible to cancel an activity by adding a so-called “shadow activity”. Both the real activity and the shadow activity are preceded by a deferred choice. 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 the token from each of its input places. 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.

□

Note that the semantics of this pattern may become ill defined if it is used in combination with multiple instances. We assume that the cancellation of an activity refers to a single instance of that activity. For the cancellation of an entire case, we assume that all instances of each activity is cancelled.

Pattern 20 (Cancel Case)
Description A case, i.e. workflow instance, is removed completely (i.e., even if parts of the

process are instantiated multiple times, all descendants are removed).
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.
Implementation

- Pattern 19 (Cancel Activity) 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 19 (Cancel Activity), many workflow management systems support the withdrawal of cases using an API which simply removes the corresponding entries from the database.

□

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 15 workflow management systems (COSA, Visual Workflow, Forté Conductor, Lotus Domino Workflow, Meteor, Mobile, MQSeries/Workflow, Staffware, Verve Workflow, I-Flow, InConcert, Changengine, SAP R/3 Workflow, Eastman, and FLOWer) based on the workflow patterns presented in this paper. We would like to point out that the common practice of choosing workflow technology before a thorough analysis of business processes in an organization may lead to a choice of a workflow product that has inadequate support for workflow patterns that could be common in this organization. In our consulting practice we have found that advanced workflow patterns as presented in this paper are frequently needed. In addition, we would like to stress that the comparison is based on the information in our possession at the end of 2001 and that we cannot guarantee the accuracy of product-specific information. However, we have found that most new versions of workflow products bring new features in areas of performance, integration approaches, new platform support, etc and feature minimal changes to the workflow modeling language which forms the core of the product. Therefore, many of the results presented in this paper will also hold for future versions of the product.

It should be noted that the goal of this section is to demonstrate that there are relevant differences between workflow products and that the set of patterns presented in this paper is a useful tool to compare these products. The goal is not to completely evaluate individual products.

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.

Staffware [Sta00] is one of the leading workflow management systems. Staffware is authored and distributed by Staffware PLC. We used Staffware 2000, which was released in the last quarter of 1999, for our evaluation. 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.

COSA [SL99] is a Petri-net-based workflow management system developed by Ley GmbH (formerly operating under the names Software Ley, COSA Solutions, and Baan). Ley GmbH is a German company based in Pullheim (Germany) and is part of Thiel Logistik AG. 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. This evaluation is based on version 3.0. 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 Choice, and Interleaved Parallel Routing are supported in a direct and graphical manner. Tokens 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.

InConcert has been established in 1996 as a Xerox fully-owned subsidiary. In 1999 it has been bought by TIBCO Software. This evaluation is based on InConcert 2000 [Tib00] (version 5.1). 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 patterns are not supported.

Eastman Software offers a variety of imaging products. Their software is used to electronically capture, share, display, fax, print, and store vital document-based information. On top of their imaging products, Eastman Software also offers a workflow management system. Enterprise Workflow 4.0, a component of the Eastman Software Enterprise Work Manager Series, provides a so-called RouteBuilder tool to design workflow processes consisting of different types of work steps [Sof98]. The following types of work steps (i.e., activity types) are supported: custom, system, archive, print, OCR, fax, transfer, program, rendezvous, split, and join. The standard semantics of a work step is an XOR-join/XOR-split semantics. The rendezvous, split, and join steps have been added to allow for parallel routing. For each join step, the user can indicate how many threads need to be synchronized. Moreover, using techniques based on the number of active parallel threads, join steps are bypassed if synchronization is not possible. This leads to constructs similar to the false-token propagation in MQSeries.

FLOWer is Pallas Athena's case handling product [Ath01]. This evaluation is based on version 2.05. FLOWer can be used for flexibly structured processes, but also supports traditional production workflow functionality. The case handling mechanisms of FLOWer solve many of the flexibility problems of traditional workflow management systems. Flexibility is guaranteed through data-driven workflows, redo and skip capabilities, and activity independent forms. FLOWer consists of a number of components: FLOWer Studio, FLOWer Case Guide, FLOWer

CFM, FLOWer Queues/Queries, FLOWer Integration Facility, and FLOWer Management Information and Case History Logging. FLOWer Studio is the graphical design environment. It is used to define processes, activities, precedences, data objects, and forms. FLOWer Case Guide is the client application which is used to handle individual cases. FLOWer queue corresponds to the worktray, worklist or in-basket of traditional WFM systems. The FLOWer queue provides a refined mechanism to look for cases satisfying specified search criteria. FLOWer CFM (ConFiguration Management) is used to define users (i.e. actors), work profiles, and authorization profiles. The profiles are used to map users onto roles. FLOWer CFM is also available at the operational level to allow for run-time flexibility. FLOWer Management Information and Case History Logging can be used to store and retrieve management information at various levels of detail. FLOWer Integration Facility provides the functionality to interface with other applications. The modeling language of FLOWer is block-structured. Blocks are named *plans* can be nested and there are five types of plans: static, dynamic, sequential, user decision and system decision. The static plan is used to specify subprocesses. The dynamic subplan is used to model multiple instances. The sequential subplan is used to model iteration. The user decision corresponds to the deferred choice and the system decision corresponds to the explicit choice. In this paper, we do not focus on the case handling facilities. We evaluate FLOWer as if it is a workflow management system. Note that the case handling capabilities may reduce the need for some of the patterns mentioned in this paper [AB01].

Domino Workflow [NEG$^+$00] is the workflow extension of the widely used groupware product Lotus Domino/Notes (Lotus/IBM). Clearly, the tight integration with the groupware product is one of the attractive features of this product. The marriage between groupware (Lotus Domino/Notes) and workflow (Domino Workflow) allows for partly structured workflows. There are various types of resource classes, e.g., person (singleton), workgroup (including inheritance and many-to-many relationships), department (only one-to-many relationships, however with inheritance), and roles. Each routing relation is of one of the following types: (1) always (for AND-split) (2) exclusive choice (for XOR-split made by the user at the end of the activity), (3) multiple choice (for OR-split made by the user after completing the activity), (4) condition (automatically evaluated on the basis of data elements), and (5) else (only taken if none of the other routing relations is activated). Each activity can serve as a join. The type of join is determined implicitly. Joins are either enabled or disabled. If a join is disabled, it serves as an XOR-join, i.e., the activity is enabled the moment one of the preceding activities completes. If the join is enabled, it continuously checks whether potentially it can receive more inputs in the future without activating itself. This way it is possible to make AND-joins or use more advanced synchronization mechanisms.

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). Our evaluation is based on the 1999 version of Meteor. 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. The Multi-merge and Arbitrary cycles patterns are supported.

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. Our evaluation is based on the 1999 version of Mobile. 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 [IBM99] is the successor of IBM's 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 move from the proprietary middleware to middleware based on the MQSeries product. Our evaluation is based on version 3.1 of the product. The workflow model consists of activities linked by transitions. Other than a decomposition block, few other special modeling constructs are 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 (note that this construct is not supported in version 3.1 of the product). 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 and cancellation patterns.

Forté Conductor [For98] is a workflow engine that is an add-on to Forté's development environment, Forté 4GL (formerly Forté Application Environment). Conductor's engine is based on experimental work performed at Digital Research and its modeling language is powerful and flexible. Forté Software has recently (in October 1999) been acquired by Sun Microsystems and subsequently became part of iPlanet E-Commerce Solutions. In late 2000 version 3.0 of the product became an integral part of iPlanet Integration Server. Our evaluation is based on version 1.0 of the product. 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 N-out-of-M join (see also Pattern 9 (Discriminator)) 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 subsequent 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.

Verve [Ver00] is a relative newcomer to the workflow market as it debuted in 1998. In late 2000 it was acquired by Versata and renamed Versata Integration Server (VIS). Our evaluation is based on version 2.1 of the product that was released just before the acquisition by Versata. What makes Verve Workflow Engine 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.

Visual WorkFlo [Fil97, Fil99] is one of the market leaders in the workflow industry. It is part of the FileNet's Panagon suite (Panagon WorkFlo Services) 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. Our evaluation is based on version 3.0 of the product. The workflow modeling language of Visual WorkFlo is highly structured and is a collection of activities and routing elements such as Branch (XOR-split), While (structured loop), Static Split (AND-split), Rendezvous (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.

Changengine [HP00] is a workflow offering from HP, the second largest computer supplier in the world. The first major version of the product, 3.0, was introduced in 1998 and it focused on high performance and support for dynamic modifications. In late 2000 the product changed its name to HP Process Manager to better convey the purpose of the product to the customers. Our evaluation is based on version 4.0, introduced in early 2000. 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.

I-Flow [Fuj99] is a workflow offering from Fujitsu that can be seen as a successor of the 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. Our evaluation is based on version 3.5 of the product, introduced in early 2000. As of the beginning of 2002 the latest version of the product is 4.1. 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.

SAP R/3 Workflow [SAP97] 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. Our evaluation is based on release 3.1 of the product. Note that SAP workflow should not be confused with EPCs (Event-driven Process Chains) found in ARIS (IDS Prof. Scheer) and in other parts of the SAP system. EPCs there are used entirely for business process modeling purposes and not for modeling executable workflows in the SAP R/3 runtime environment. SAP R/3 Workflow imposes a number of restrictions on the use of EPCs. EPCs that are used for workflow modeling 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.

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. If a product directly supports the pattern through one of its constructs, it is rated +. If the pattern is not *directly* supported, it is rated +/-. Any solution which results in spaghetti diagrams or coding, is considered as giving no direct support and is rated -.

Note that a pattern is only supported directly if there is a feature provided by the graphical interface of the tool (i.e., not in some scripting language) which supports the construct without resorting to any of solutions mentioned in the implementation part of the pattern. For example, Pattern 6 (Multi-choice) can be realized using a network of AND/XOR-splits. However, this does not mean that any workflow systems supporting patterns 2 and 4 directly supports Pattern 6. Consider for example Figure 1. A system that allows for a representation similar to the one shown in Figure 1 on the left (Workflow A) offers direct support. The other two representations (Workflow B and Workflow C) do not correspond to direct support.

If a pattern is not directly supported or even not supported at all by the workflow management system, this does not imply that it is impossible to realize the functionality. Again consider Pattern 6 (Multi-choice). It is possible to realize this pattern by creating a network of AND/XOR-splits (i.e., using patterns 2 and 4). However, if no specific support is given for Pattern 6, it is rated -. An alternative rating could have been based on “implementation effort” rather than direct support (+), partial/indirect support (+/-) or no support (-). However, any attempt to quantify this implementation effort is subjective and depends on the expertise of the designer.

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

[1]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							
	Staffware	COSA	InConcert	Eastman	FLOWer	Domino	Meteor	Mobile[1]
1 (seq)	+	+	+	+	+	+	+	+
2 (par-spl)	+	+	+	+	+	+	+	+
3 (synch)	+	+	+	+	+	+	+	+
4 (ex-ch)	+	+	+/-	+	+	+	+	+
5 (simple-m)	+	+	+/-	+	+	+	+	+
6 (m-choice)	-	+	+/-	+/-	-	+	+	+
7 (sync-m)	-	+/-	+	+	-	+	-	-
8 (multi-m)	-	-	-	+	+/-	+/-	+	-
9 (disc)	-	-	-	+	+/-	-	+/-	+
10 (arb-c)	+	+	-	+	-	+	+	-
11 (impl-t)	+	-	+	+	-	+	-	-
12 (mi-no-s)	-	+/-	-	+	+	+/-	+	-
13 (mi-dt)	+	+	+	+	+	+	+	+
14 (mi-rt)	-	-	-	-	+	-	-	-
15 (mi-no)	-	-	-	-	+	-	-	-
16 (def-c)	-	+	-	-	+/-	-	-	-
17 (int-par)	-	+	-	-	+/-	-	-	+
18 (milest)	-	+	-	-	+/-	-	-	-
19 (can-a)	+	+	-	-	+/-	-	-	-
20 (can-c)	-	-	-	-	+/-	+	-	-

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

pattern	product						
	MQSeries	Forté	Verve	Vis. WF	Changeng.	I-Flow	SAP/R3
1 (seq)	+	+	+	+	+	+	+
2 (par-spl)	+	+	+	+	+	+	+
3 (synch)	+	+	+	+	+	+	+
4 (ex-ch)	+	+	+	+	+	+	+
5 (simple-m)	+	+	+	+	+	+	+
6 (m-choice)	+	+	+	+	+	+	+
7 (sync-m)	+	-	-	-	-	-	-
8 (multi-m)	-	+	+	-	-	-	-
9 (disc)	-	+	+	-	+	-	+
10 (arb-c)	-	+	+	+/-	+	+	-
11 (impl-t)	+	-	-	-	-	-	-
12 (mi-no-s)	-	+	+	+	-	+	-
13 (mi-dt)	+	+	+	+	+	+	+
14 (mi-rt)	-	-	-	-	-	-	+/-
15 (mi-no)	-	-	-	-	-	-	-
16 (def-c)	-	-	-	-	-	-	-
17 (int-par)	-	-	-	-	-	-	-
18 (milest)	-	-	-	-	-	-	-
19 (can-a)	-	-	-	-	-	-	+
20 (can-c)	-	+	+	-	+	-	+

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

Please apply the results summarized in tables 1 and 2 with care. First of all, the organization selecting a workflow management system should focus on the patterns most relevant for the workflow processes at hand. Since support for the more advanced patterns is limited, one should focus on the patterns most needed. Second, the fact that a pattern is not directly supported by a product does not imply that it is not possible to support the construct at all. As indicated throughout the paper, many patterns can be supported indirectly through mixtures of more basic patterns and coding. Third, the patterns reported in this paper only focus on the process perspective (i.e., control flow or routing). The other perspectives (e.g., organizational modeling) should also be taken into account. Moreover, additional features of the tool may reduce the need for certain routing constructs. For example, Lotus Domino Workflow is embedded in a complete groupware system which reduces the need for certain constructs. Another example is the case handling tool FLOWer. The case-handling paradigm allows for implicit routing which reduces the need some of the constructs (e.g., arbitrary loops and advanced synchronization constructs).

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.

Recently we evaluated five workflow projects conducted by ATOS/Origin (Utrecht, The Netherlands) to get quantitative data about the frequency of patterns [VO01, VO02]. Each of these projects involved multiple processes with processes ranging from dozens of activities to hundreds of activities. The projects used three of the workflow products mentioned in this paper (Eastman, Staffware, and Domino Workflow) and the results obtained show that in most of the projects evaluated there is a strong need for the more advanced patterns presented in this paper. Empirical findings show that in many projects workflow designers are forced to adapt the process or need to resort to spaghetti-like diagrams or coding. Another interesting observation is that there seems to be a correlation between the patterns being used and the workflow product being deployed, e.g., the processes developed in projects using Staffware have much more parallelism than the processes developed in projects using Eastman. Further research is needed to truly understand the influence of the workflow product on the processes being supported.

Acknowledgements. We would like to thank the anonymous referees for their useful comments that helped to improve this paper. We also thank the vendors and consultants that cooperated by reviewing the results. In particular, we thank Kristiaan de Vries, Ton Pijpers, Hajo Reijers, Jaap Rigter, Eric Verbeek, Dennis Smit, Paul Berens, Dolf Grunbauer, and Oscar Ommert.

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.

[AB01] W.M.P. van der Aalst and P.J.S. Berens. Beyond Workflow Management: Product-Driven Case Handling. In S. Ellis, T. Rodden, and I. Zigurs, editors, *International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2001)*, pages 42–51. ACM Press, New York, 2001.

[AH02] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT press, Cambridge, MA, 2002.

[AHKB] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns Home Page. http://www.tm.tue.nl/it/research/patterns/.

[AHKB00a] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, Berlin, 2000.

[AHKB00b] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, 2000.

[AM00] A. Agostini and G. De Michelis. Improving Flexibility of Workflow Management Systems. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 218–234. Springer-Verlag, Berlin, 2000.

[Ath01] Pallas Athena. *Flower User Manual.* Pallas Athena BV, Apeldoorn, The Netherlands, 2001.

[BW99] P. Barthelmess and J. Wainer. Enhancing workflow systems expressive power. (unpublished), 1999.

[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.

[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.

[Fil97] FileNet. *Visual WorkFlo Design Guide.* FileNet Corporation, Costa Mesa, CA, USA, 1997.

[Fil99] FileNet. *Panagon Visual WorkFlo Architecture.* FileNet Corporation, Costa Mesa, CA, USA, 1999.

[For98] Forté. *Forté Conductor Process Development Guide.* Forté Software, Inc, Oakland, CA, USA, 1998.

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

[Fuj99] Fujitsu. *i-Flow Developers Guide.* Fujitsu Software Corporation, San Jose, CA, USA, 1999.

[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.

[Hir01] A. Hirnschall. Patterns Discusion in the Context of Co-flow, CONDIS Workflow Management System (in German). Technical report, University of Linz, Austria, 2001.

[HP00] HP. *HP Changengine Process Design Guide.* Hewlett-Packard Company, Palo Alto, CA, USA, 2000.

[IBM99] IBM. *IBM MQSeries Workflow - Getting Started With Buildtime.* IBM Deutschland Entwicklung GmbH, Boeblingen, Germany, 1999.

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

[KAV02] A. Kumar, W.M.P. van der Aalst, and H.M.W. Verbeek. Dynamic Work Distribution in Workflow Management Systems: How to Balance Quality and Performance? *Journal of Management Information Systems*, 18(3):157–193, 2002.

[KDB00] M. Klein, C. Dellarocas, and A. Bernstein, editors. *Adaptive Workflow Systems*, volume 9 of *Special issue of the journal of Computer Supported Cooperative Work*, 2000.

[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.

[Lav00] H. Lavana. *A Universally Configurable Architecture for Taskflow-Oriented Design of a Distributed Collaborative Computing Environment* . PhD thesis, Department of Computer Science, North Carolina State University, Raleigh, NC, USA, 2000.

[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.

[MB97] G. Meszaros and K. Brown. A Pattern Language for Workflow Systems. In *Proceedings of the 4th Pattern Languages of Programming Conference*, Washington University Technical Report 97-34 (WUCS-97-34), 1997.

[NEG$^+$00] S.P. Nielsen, C. Easthope, P. Gosselink, K. Gutsze, and J. Roele. *Using Lotus Domino Workflow 2.0, Redbook SG24-5963-00*. IBM, Poughkeepsie, USA, 2000.

[Pro98] Promatis. *Income Workflow User Manual*. Promatis GmbH, Karlsbad, Germany, 1998.

[RR98] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

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

[SAA99] A.P. Sheth, W.M.P. van der Aalst, and I.B. Arpinar. Processes Driving the Networked Economy: ProcessPortals, ProcessVortex, and Dynamically Trading Processes. *IEEE Concurrency*, 7(3):18–31, 1999.

[SAP97] SAP. *WF SAP Business Workflow*. SAP AG, Walldorf, Germany, 1997.

[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.

[SL99] Software-Ley. *COSA 3.0 User Manual*. Software-Ley GmbH, Pullheim, Germany, 1999.

[Sof98] Eastman Software. *RouteBuilder Tool User's Guide*. Eastman Software, Inc, Billerica, MA, USA, 1998.

[Sta00] Staffware. *Staffware 2000 / GWD User Manual*. Staffware plc, Berkshire, United Kingdom, 2000.

[Tib00] Tibco. *TIB/InConcert Process Designer User's Guide*. Tibco Software Inc., Palo Alto, CA, USA, 2000.

[Ver00] Verve. *Verve Component Workflow Engine Concepts*. Verve, Inc., San Francisco, CA, USA, 2000.

[VO01] K. de Vries and O. Ommert. Advanced Workflow Patterns in Practise (1): Experiences Based on Pension Processing (in Dutch). *Business Process Magazine*, 7(6):15–18, 2001.

[VO02] K. de Vries and O. Ommert. Advanced Workflow Patterns in Practice (2): Experiences Based on Judicial Processes (in Dutch). *Business Process Magazine*, 8(1):20–23, 2002.

[WAH00] H. Weigand, A. de Moor, and W.J. van den Heuvel. Supporting the Evolution of Workflow Patterns for Virtual Communities. *Electronic Markets*, 10(4):264–271, 2000.

[WFM99] WFMC. Workflow Management Coalition Terminology & Glossary, Document Number WFMC-TC-1011, Document Status - Issue 3.0, February. Technical report, Workflow Management Coalition, Brussels, 1999.

A Staffware

pattern	score	motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported through a step with multiple output arcs.
3 (synch)	+	Supported through a so-called wait step. There is an asymmetry which results in unexpected behavior if multiple synchronizations take place concurrently.
4 (ex-ch)	+	Supported through a so-called decision. The decision is binary.
5 (simple-m)	+	Supported through a step with multiple input arcs.
6 (m-choice)	-	Wait steps are binary and correspond to exclusive OR-splits.
7 (sync-m)	-	No support.
8 (multi-m)	-	No support. Note that it is not possible to enable a step twice. The second thread will cancel the first one.
9 (disc)	-	No support.
10 (arb-c)	+	There are some syntactical limitations. However, it is possible to have multiple intertwined cycles.
11 (impl-t)	+	Directly supported. The workflow instance terminates if all of the corresponding branches have terminated.
12 (mi-no-s)	-	Only through a graphical enumeration of the number of instances.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	-	No support.
15 (mi-no)	-	No support.
16 (def-c)	-	There is no state concept. The only way to support this pattern is through a parallel split and a withdraw. However, this solution is complex and not safe.
17 (int-par)	-	No support.
18 (milest)	-	No support.
19 (can-a)	+	Directly supported.
20 (can-c)	-	No support, only through API.

B COSA

pattern	score	motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	If no conditions specified, AND-split semantics.
3 (synch)	+	If no conditions specified, AND-join semantics.
4 (ex-ch)	+	Through conditions on output arcs of activities.
5 (simple-m)	+	Through places (named conditions in COSA).
6 (m-choice)	+	Through conditions on output arcs of activities.
7 (sync-m)	+/-	The condition on an input arc can be used to state that a token is not needed, i.e., the standard condition is weakened to avoid full synchronization.
8 (multi-m)	-	Only safe Petri net diagrams can be used.
9 (disc)	-	The discriminator can be modeled by using true conditions in input arcs and extending the network. Unfortunately, the resulting diagram is too complex.
10 (arb-c)	+	Supported. Any graph structure is allowed.
11 (impl-t)	-	Not supported, explicit termination is needed.
12 (mi-no-s)	+/-	COSA has a three level workflow model, i.e., workflow, flow, and activity. Flows (i.e., workflow instances) can be grouped in one workflow and share information. This combined with a trigger mechanism to create new flows is a solution.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	-	Not supported.
15 (mi-no)	-	Not supported.
16 (def-c)	+	Directly supported through places.
17 (int-par)	+	Directly supported through places and also an optional setting of the workflow engine.
18 (milest)	+	Directly supported through places.
19 (can-a)	+	Directly supported by removing tokens from input places.
20 (can-c)	-	Only supported through an API.

C InConcert

pattern	score	motivation
1 (seq)	+	Directly supported through arrows representing explicit dependencies.
2 (par-spl)	+	Directly supported through multiple output arcs.
3 (synch)	+	Directly supported through multiple input arcs.
4 (ex-ch)	+/-	There is only conditional routing through so-called “perform conditions”. These conditions determine whether an activity or subprocess should be executed. This simplifies the modeling by end users but results in only partial support for the pattern.
5 (simple-m)	+/-	Only indirectly supported through perform conditions.
6 (m-choice)	+/-	Not directly supported. However, it is possible to use perform conditions in the top-level process of each branch. Therefore, an intermediate rating is given.
7 (sync-m)	+	Any split is an AND-split and any join is an AND-join. Therefore, the pattern is supported through perform conditions.
8 (multi-m)	-	Since every join in InConcert is an AND-join, there is no merge and no way to start a branch twice.
9 (disc)	-	Since every join is an AND-join, there is no way to continue before all branches have completed.
10 (arb-c)	-	There are no loops.
11 (impl-t)	+	Directly supported: any acyclic graph will do.
12 (mi-no-s)	-	Not supported.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	-	Not supported.
15 (mi-no)	-	Not supported.
16 (def-c)	-	There is no state concept. It is also not possible to withdraw an activity. However, the perform condition can be set to bypass activities in other branches.
17 (int-par)	-	Not supported: the perform condition does not block processing (instead activities are skipped).
18 (milest)	-	Not supported: there is no state concept.
19 (can-a)	-	Users can cancel activities. However, there is no way to structure this and have one activity cancel another one.
20 (can-c)	-	Users can cancel cases given proper authorization. However, there is no way to do this automatically.

D Eastman

pattern	score	motivation
1 (seq)	+	Directly supported through arcs connecting worksteps.
2 (par-spl)	+	Supported by split worksteps.
3 (synch)	+	Supported by join worksteps.
4 (ex-ch)	+	The semantics of a "normal" workstep is XOR-split.
5 (simple-m)	+	The semantics of a "normal" workstep is XOR-join.
6 (m-choice)	+/-	A small network consisting of a split workstep and an additional workstep for each branch is needed. However, it is also possible to use preprocessing conditions in the first workstep of each branch. Therefore, an intermediate rating is given.
7 (sync-m)	+	Supported.
8 (multi-m)	+	Directly supported.
9 (disc)	+	Supported. However, it works only partially. Running instances/threads can activate the construct before it finishes.
10 (arb-c)	+	Directly supported.
11 (impl-t)	+	Directly supported.
12 (mi-no-s)	+	Multiple threads can be created and follow the same path in the process.
13 (mi-dt)	+	Multiple threads can be created and follow the same path in the process. A join workstep can have an attribute which specifies how many instances should be joined.
14 (mi-rt)	-	The join workstep attribute which specifies how many instances should be joined, is fixed.
15 (mi-no)	-	The join workstep attribute which specifies how many instances should be joined, is fixed.
16 (def-c)	-	No states.
17 (int-par)	-	There are no states nor semaphore-like facilities.
18 (milest)	-	Not supported.
19 (can-a)	-	Not supported. There is a "remove from workflow" command and a delete workstep. Both commands can be used to terminate an instance/thread. However, one tread can only terminate itself and not other threads.
20 (can-c)	-	Not supported.

E FLOWer

FLOWer is a so-called case-handling system. Since it is mainly data-driven, an evaluation based on the patterns is not straightforward. Therefore, we provide a more detailed argumentation.

pattern	score	motivation
1 (seq)	+	Directly supported through arcs connecting plan elements.
2 (par-spl)	+	Nodes in a subplan (static, dynamic, and sequential) have an AND-split semantics.
3 (synch)	+	Nodes in a subplan (static, dynamic, and sequential) have an AND-join semantics.
4 (ex-ch)	+	Supported through the plan type system decision (based on data) and the plan type user decision (based on a user selection on the wavefront).
5 (simple-m)	+	Supported by the end nodes of the plan type system decision and the plan type user decision.
6 (m-choice)	-	Not supported: the decision plan types only allow for a 1-out-of-m selection.
7 (sync-m)	-	Not supported.
8 (multi-m)	+/-	It is possible the have multiple concurrent threads using dynamic subplans. Therefore, there is partial support for the pattern. However, since all networks are highly structured, it is not possible to have an AND-split/XOR-join type of situation.
9 (disc)	+/-	The discriminator is not directly supported. However, dynamic subplans can have a so-called auto complete condition. This condition allows for emulating constructs such as the discriminator and the n-out-of-m join.
10 (arb-c)	-	Not supported. In fact there are no loops and the language is block structured with AND-type of blocks and XOR-type of blocks. Iteration is achieved through the sequential subplan and the redo role.
11 (impl-t)	-	Not supported: every plan has a unique start node and end node.
12 (mi-no-s)	+	Directly supported through dynamic subplans.
13 (mi-dt)	+	Directly supported through dynamic subplans and supported through a combination of splits and joins.
14 (mi-rt)	+	Directly supported through dynamic subplans. One can specify a variable number of instances.
15 (mi-no)	+	Directly supported through dynamic subplans. It is possible to create new instances, while executing.
16 (def-c)	+/-	There is no explicit notion of states. The plan type user decision (based on a user selection on the wavefront) can solve the implicit choice in some cases. The plan type system decision can solve the implicit choice in some other cases. Note that a system decision blocks until at least one of its conditions is true. This way "race conditions" based on time or external triggers are possible. In the latter case triggering is handled through data-dependencies rather than explicit control-flow dependencies. Moreover, mixtures of system and user decisions are problematic.

pattern	score	motivation
17 (int-par)	+/-	Due to the case metaphor there is just one actor working on the case. Therefore, there is no true concurrency and any parallel routing is interleaved. Since true concurrency is not possible, an intermediate rating is given.
18 (milest)	+/-	There is no direct support for milestones since there is no notion of states. However, in all situations data dependencies can be used to emulate the construct. Simply introduce for each state (i.e., place in Petri-net terms) a data element.
19 (can-a)	+/-	It is possible to skip or redo activities. However, it is not possible to withdraw an activity in one branch triggered by an activity in another branch. Skip and redo are explicit user actions. Therefore, they provide only partial support.
20 (can-c)	+/-	It is possible to skip or redo an entire plan. However, skip and redo actions are always explicit user actions. Therefore, they provide only partial support. Note that using a date element named cancel and using this data element as a precondition for every activity in the flow it is possible to block a case. Although this is an elegant solution, it is still considered to be indirect.

F Domino Workflow

pattern	score	motivation
1 (seq)	+	Directly supported through routing relations connecting activities.
2 (par-spl)	+	By making each output routing relation of type “always”.
3 (synch)	+	By setting the join attribute to “enabled”.
4 (ex-ch)	+	By selecting either “condition” or “multiple choice” on output relations.
5 (simple-m)	+	By selecting “disable join”.
6 (m-choice)	+	By selecting either “condition” or “multiple choice” on output relations.
7 (sync-m)	+	Supported. Select “enable join”. The AND-join will wait as long as something may arrive.
8 (multi-m)	+/-	Partially supported. Select “enable join”. If instances meet, they are merged. Therefore, no full support. Note that even if the individual subcases are merged, they are still visible in the content of the case.
9 (disc)	-	Not directly supported. Only through scripting.
10 (arb-c)	+	Supported, only the self-loop is excluded.
11 (impl-t)	+	Supported, there may be multiple end nodes.
12 (mi-no-s)	+/-	No graphical support for multiple instances. However, the scripting language of Domino allows for the creation of parallel instances. Moreover, by selecting “disable join” it is possible to deal with multiple instances of sequential parts of the process.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	-	No graphical support for multiple instances. However, the scripting language of Domino allows for counters waiting for all instances to terminate.
15 (mi-no)	-	No direct support.
16 (def-c)	-	No states.
17 (int-par)	-	For parallel routing, the case is split into identical copies which are merged/joined at the AND-join. Therefore, only true parallelism is supported.
18 (milest)	-	No states.
19 (can-a)	-	Not possible. Note that activities can be rerouted.
20 (can-c)	+	The activity owner can issue a “request to cancel job”. The job owner decides on the actual deletion of the job.

G Meteor

pattern	score	motivation
1 (seq)	+	Directly supported. Tasks (i.e., activities) in Meteor have a “to” clause to model causal dependencies.
2 (par-spl)	+	Directly supported. The semantics of multiple “to” clauses inside a task is AND-split.
3 (synch)	+	Directly supported. Tasks have a “and_or” clause which allow for synchronization.
4 (ex-ch)	+	Directly supported. The “to” clauses of a task can have a condition associated to them.
5 (simple-m)	+	Directly supported. Tasks have a “and_or” clause which allow for XOR-join semantics.
6 (m-choice)	+	Supported through conditional “to” clauses.
7 (sync-m)	-	Not supported.
8 (multi-m)	+	Supported. Multi-threaded workflows are possible.
9 (disc)	+/-	Using the “and_or” clause it is possible to conditionally synchronize. This mechanism can also be used for the n-out-of-m. However, there is no garbage collection. A 2-to-out-10 construct may lead to 5 instances of subsequent parts of the process.
10 (arb-c)	+	Directly supported.
11 (impl-t)	-	Not supported. One unique final task (stop task).
12 (mi-no-s)	+	Supported. Multi-threaded workflows are possible.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	-	Not supported.
15 (mi-no)	-	Not supported.
16 (def-c)	-	Not supported. There is no state concept.
17 (int-par)	-	The architecture is highly distributed and parallel branches are completely independent. There is no mechanism to interleave these branches.
18 (milest)	-	Not supported. There is no state concept.
19 (can-a)	-	Not directly supported.
20 (can-c)	-	Not directly supported.

H Mobile

Mobile is extensible. Therefore, it is possible to add new constructs and, in principle, Mobile could support any of the patterns. In this table, we only list the constructs already available.

pattern	score	motivation
1 (seq)	+	Supported through the “sequence” construct.
2 (par-spl)	+	Supported through the “parallel” construct.
3 (synch)	+	Supported through the “parallel” construct.
4 (ex-ch)	+	Supported through the “if_then_else” construct.
5 (simple-m)	+	Supported through the “if_then_else” construct.
6 (m-choice)	+	Supported through the “case” construct.
7 (sync-m)	-	Not directly supported.
8 (multi-m)	-	Not directly supported.
9 (disc)	+	Supported through the “n-out-of-m” construct.
10 (arb-c)	-	Not supported. Mobile is a block structured language with an underlying textual language. Therefore, it is not possible to have arbitrary cycles.
11 (impl-t)	-	Not directly supported.
12 (mi-no-s)	-	Not directly supported.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	-	Not directly supported.
15 (mi-no)	-	Not directly supported.
16 (def-c)	-	Not directly supported.
17 (int-par)	+	Supported through the “reihung/anysequence” construct.
18 (milest)	-	Not directly supported.
19 (can-a)	-	Not directly supported.
20 (can-c)	-	No directly support.

I MQSeries Workflow

pattern	score	motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Directly supported through multiple outgoing transitions of an activity.
3 (synch)	+	Directly supported.
4 (ex-ch)	+	Supported through the use of exclusive conditions on transitions.
5 (simple-m)	+	Directly supported.
6 (m-choice)	+	Supported through the use of non-exclusive conditions on transitions.
7 (sync-m)	+	Directly supported.
8 (multi-m)	-	Not supported.
9 (disc)	-	Not supported.
10 (arb-c)	-	Not supported.
11 (impl-t)	+	Directly supported.
12 (mi-no-s)	-	Not supported.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	-	Used to be supported through a special construct called a "bundle". Sadly, the bundle construct is curiously missing in the latest version of the product.
15 (mi-no)	-	Not supported.
16 (def-c)	-	Not supported.
17 (int-par)	-	Not supported.
18 (milest)	-	Not supported.
19 (can-a)	-	Not supported.
20 (can-c)	-	Not supported.

J Forté Conductor

pattern	score	motivation
1 (seq)	+	Directly supported using activity routers.
2 (par-spl)	+	Supported by using multiple outgoing routers.
3 (synch)	+	Supported by specifying a special trigger condition for an activity with multiple incoming routers.
4 (ex-ch)	+	Supported by using multiple outgoing routers with router conditions.
5 (simple-m)	+	Supported by specifying a special trigger condition for an activity with multiple incoming routers.
6 (m-choice)	+	Supported by specifying a special trigger condition for an activity with multiple incoming routers.
7 (sync-m)	-	Not supported.
8 (multi-m)	+	Supported by using multiple outgoing routers with router conditions.
9 (disc)	+	Supported through the use of custom condition and process variables.
10 (arb-c)	+	Directly supported.
11 (impl-t)	-	Not supported.
12 (mi-no-s)	+	Directly supported.
13 (mi-dt)	+	Directly supported.
14 (mi-rt)	-	Not supported.
15 (mi-no)	-	Not supported.
16 (def-c)	-	No concept of a state.
17 (int-par)	-	Not supported.
18 (milest)	-	Not supported.
19 (can-a)	-	Not supported.
20 (can-c)	+	A final, terminating, task could be use to terminate a process instance.

K Verve

pattern	score	motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Directly supported.
3 (synch)	+	Supported through a “Synchronizer” construct.
4 (ex-ch)	+	Supported through exclusive conditions on arcs.
5 (simple-m)	+	Directly supported.
6 (m-choice)	+	Supported through non-exclusive conditions on arcs.
7 (sync-m)	-	Not supported.
8 (multi-m)	+	Directly supported.
9 (disc)	+	Supported through a “Discriminator” construct.
10 (arb-c)	+	Directly supported.
11 (impl-t)	-	Not supported.
12 (mi-no-s)	+	Directly supported.
13 (mi-dt)	+	Supported.
14 (mi-rt)	-	Not supported.
15 (mi-no)	-	Not supported.
16 (def-c)	-	Not supported.
17 (int-par)	-	Not supported.
18 (milest)	-	Not supported.
19 (can-a)	-	Not supported.
20 (can-c)	+	Supported through a final activity.

L Visual WorkFlo

pattern	score	motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Directly supported.
3 (synch)	+	Supported using “Rendezvous” construct.
4 (ex-ch)	+	Supported using “Branch” construct.
5 (simple-m)	+	Directly supported.
6 (m-choice)	+	Supported through a combination of other constructs.
7 (sync-m)	-	Not supported.
8 (multi-m)	-	Not supported.
9 (disc)	-	Not supported.
10 (arb-c)	+/-	Supported to some extent through the use of “Goto” construct. Note though that the standard loop in Visual WorkFlo is fully structured.
11 (impl-t)	-	Not supported.
12 (mi-no-s)	+	Supported through the “Release” construct.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	-	Not supported.
15 (mi-no)	-	Not supported.
16 (def-c)	-	Not supported.
17 (int-par)	-	Not supported.
18 (milest)	-	Not supported.
19 (can-a)	-	Not supported.
20 (can-c)	-	Not supported.

M Changengine

pattern	score	motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported through a Route Node with more than one outgoing arcs.
3 (synch)	+	Supported through a Route Node with more than one incoming arcs.
4 (ex-ch)	+	Supported through a Route Node with more than one outgoing arcs.
5 (simple-m)	+	Supported through a Route Node with more than one incoming arcs.
6 (m-choice)	+	Supported through a Route Node with more than one outgoing arcs.
7 (sync-m)	-	Not supported.
8 (multi-m)	-	Not supported.
9 (disc)	+	Supported through a Route Node with more than one incoming arcs.
10 (arb-c)	+	Directly supported.
11 (impl-t)	-	Not supported.
12 (mi-no-s)	-	Not supported.
13 (mi-dt)	+	Directly supported.
14 (mi-rt)	-	Not supported.
15 (mi-no)	-	Not supported.
16 (def-c)	-	Not supported.
17 (int-par)	-	Not supported.
18 (milest)	-	Not supported.
19 (can-a)	-	Not supported.
20 (can-c)	+	Supported through “Abort Nodes”.

N I-Flow

pattern	score	motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Directly supported.
3 (synch)	+	Supported through an AND node.
4 (ex-ch)	+	Supported through a Conditional Node.
5 (simple-m)	+	Directly supported.
6 (m-choice)	+	Supported through a combination of other constructs.
7 (sync-m)	-	Not supported.
8 (multi-m)	-	Not supported (no concurrent multiple instances are permitted).
9 (disc)	-	Not supported.
10 (arb-c)	+	Supported.
11 (impl-t)	-	Not supported.
12 (mi-no-s)	+	Supported through chained-process node.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	-	Not supported.
15 (mi-no)	-	Not supported.
16 (def-c)	-	Not supported.
17 (int-par)	-	Not supported.
18 (milest)	-	Not supported.
19 (can-a)	-	Not supported.
20 (can-c)	-	Not supported.

O SAP/R3 Workflow

pattern	score	motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported by a “fork” construct.
3 (synch)	+	Supported by a “join” construct.
4 (ex-ch)	+	Supported by a “condition” construct.
5 (simple-m)	+	Directly supported.
6 (m-choice)	+	Supported by a “condition” construct.
7 (sync-m)	-	Not supported.
8 (multi-m)	-	Not supported.
9 (disc)	+	Supported by a “join” construct. With SAP one can specify the number of incoming transitions to be waited for in a “fork” construct. The processing continues once the specified number of paths is completed. Note though that the remaining, unfinished, paths are terminated.
10 (arb-c)	-	Not supported.
11 (impl-t)	-	Not supported.
12 (mi-no-s)	-	Not supported.
13 (mi-dt)	+	Supported through a combination of splits and joins.
14 (mi-rt)	+/-	Supported through the use of the so-called “table-driven dynamic parallel processing”. This is a very implicit technique for specifying the desired behavior of this pattern.
15 (mi-no)	-	Not supported.
16 (def-c)	-	Not supported.
17 (int-par)	-	Not supported.
18 (milest)	-	Not supported.
19 (can-a)	+	Supported through a “workflow control” construct.
20 (can-c)	+	Supported through a “workflow control” construct.

Contents