Advanced Workflow Patterns

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

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

Abstract

Conventional workflow functionality like task sequencing, split parallelism, join synchronisation and iteration have proven effective for business process automation and have widespread support in current workflow products. One the other hand, newer requirements for workflows are encountered in practice, opening grave uncertainties about the extensions for current languages. Different concepts, although outwardly appearing to be more or less the same, are based on different paradigms, have fundamentally different semantics and different levels of applicability - more specialized for modelling or more generalized for workflow engine posit. By way of developmental insight of *new* requirements, we define workflow patterns which are described imperatively but independently of current workflow languages. Through a detailed insight into a number of commercially available workflow management systems we highlight the semantic difficulties and implementations in the current state of the art.

1 Introduction

Background

Workflow technology continues to be subjected to on-going development in its traditional application areas of business process modelling and business process coordination, and now in

*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.

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, EN93, GHS95, JB96, Kou95, Law97, Sch96, WFM96, DKTS98]).

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

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

Clearly, the control flow perspective provides an essential insight into a workflow specification's effectiveness. The data flow perspective rests on it, while the organizational and operational perspectives are ancillary. If workflow specifications are to be extended to meet newer processing requirements, control flow constructors require a fundamental insight and analysis. Currently, most workflow languages support the basic constructs of sequence, iteration, splits (AND and OR) and joins (AND and OR) - see [Law97]. However, the interpretation of even these basic constructs is not uniform and it is often unclear how more complex requirements could be supported. Indeed, vendors are afforded the opportunity to recommend implementation level "hacks" such as database triggers and application event handling. The result is that neither workflow specifications or clean insight into newer requirements is advanced.

Problem

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

The challenge, which we undertake in this paper, is to understand how complex requirements can be addressed in the current state of the art. These requirements, 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 new requirements for workflow languages through workflow *patterns*. As described in [RZ96], a pattern "is the abstraction from concrete form which keeps recurring in specific non-arbitrary contexts". Gamma et al. [GHJV95] first catalogued systematically some 23 design patterns which describe the smallest recurring interactions in object-oriented systems. The design patterns, as such, provided independence from the implementation technology and at the same time independence from the essential requirements of the domain that they were attempting to address (see also e.g. [Fow97]).

For our purpose, patterns address business requirements in an imperative workflow style expression, but are removed from specific workflow languages. Thus they do not claim to be the only way of addressing the business requirements. Nor are they "alienated" from the workflow approach, thus allowing a potential mapping to be positioned closely to different languages and implementation solutions. Along the lines of [GHJV95], patterns are described through: conditions that should hold for the pattern to be applicable; examples of business situations; problems, typically semantic problems, of realization in current languages; and implementation solutions.

The rest of the paper describes a number of workflow patterns addressing what we believe identifies complex workflow functionality. It will be assumed throughout that the reader is familiar with the basic functionality of current workflows: sequence, splits (OR and AND),

joins (OR and AND) and iteration. Where formative or clarative, reference to current workflow products will be made. In Appendix A, we provide a brief description of the workflow products referred to in this paper.

2 Advanced Synchronisation Patterns

In most workflow engines two basic forms of synchronisation are supported, AND-join and OR-join. Although the actual semantics of these constructs differ from system to system, it can be safely assumed that the intention of the AND-join is to synchronise two (or more) concurrent threads, whereas the intention of the OR-join is to merge two threads into one with the (implicit) assumption that only one thread will be active during run-time.

Many different business scenarios require more advanced synchronisation patterns. In this section we will present the ones that we believe are most commonly encountered, i.e. synchronising merge and n-out-of-m join.

Pattern 1 (Synchronising Merge)
Description A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronisation of the active threads needs to take place. If only one path is taken, the alternative branches should reconverge without synchronisation.
Synonyms Synchronising join
Examples

- After executing the activity *evaluate_damage* the activity *contact_fire_department* or the activity *contact_insurance_company* is executed. However, it is also possible that both need to be executed. After either or both of these activities have been completed, the activity *submit report* needs to be performed (exactly once).

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

- The two workflow engines known to the authors that provide a straightforward construct for the realisation of this pattern are MQSeries/Workflow and InConcert. As noted earlier, if a synchronising 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 synchronisation should take place. MQSeries/Workflow works around that problem by passing a False token for each transition that evaluates to False and a True token for each transition that evaluates to True. The merge will wait until it receives tokens from each incoming transition. InConcert does not use a False token concept. Instead it passes a token through every transition in a graph. This token may or may not enable

the execution of an activity depending on the entry condition. This way every activity having more than one incoming transition can expect that it will receive a token from each one of them, thus deadlock cannot occur. The careful reader may note that these evaluation strategies require that the workflow process does not contain cycles.

- In all other workflow engines the implementation of the synchronising merge is not straightforward. The common design pattern is to avoid the explicit use of the OR-split that may trigger more than one outgoing transition and implement it as a combination of AND-splits and OR-splits that guarantee to trigger only one of the outgoing transitions (we will call such splits XOR-splits for the remaining of this paper). This way we can easily synchronise corresponding branches by using AND-join and OR-join constructs.

□

The next pattern deals with more complex form of synchronisation when the join may want to wait for a given number of incoming transitions to fire and ignore any remaining transitions that may fire later.

Pattern 2 (N-out-of-M Join)
Description The N-out-of-M Join is a point in a workflow process where M parallel paths converge into one. The subsequent activity should be activated once N paths become active. Activation of all remaining paths should be ignored. Once all incoming branches "fire", the join resets itself so that it can fire again.
Synonyms Partial join (cf. [CCPP95]), discriminator, custom join.
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.

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

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

Problem Clearly neither standard OR-join nor AND-join can be used to implement this pattern. As mentioned before, different products implement the semantics of the standard OR-join when both incoming branches get activated differently, but they typically fall into two categories. Products that support multiple concurrent instances (Verve, Forté) will activate the activity that follows the OR-join twice. Other products (e.g. Staffware, HP Changengine, I-Flow) will not generate the second instance of an activity if the first instance is still active. However, this does not provide the solution for the join since once the first instance of the

activity finishes, the second instance will be created. These products simply do not allow two separate instances to be running at the same time.

Solutions

- There is a special construct that implements the 1-out-of-M join semantics in Verve, called *discriminator*. By combining discriminator with the standard AND-split and AND-join constructs it is possible to express N-out-of-M join semantics. An example of a 2-out-of-3 join is shown in Figure 1.

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

- In all other workflow engines the N-out-of-M join semantics is hard or impossible to implement. The common design pattern (shown here for 1-out-of-2 join) is to try to delete outstanding workitems if the workflow product allows to do so. Once the first instance of the activity following the join is created, the activities of the incoming branch that still has not been completed can be cancelled. This way the second instance of the activity following the join will not be created. This pattern is shown in the example of Figure 2. The problem with this solution is that if activities B and C are performed concurrently, activity D may still end-up being executed twice. Moreover, the original semantics of the join is to allow both B and C to finish. In this solution either B or C will get cancelled.

□

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 modelling point of view and tend to restrict the specification freedom of the business analyst. As a result, business analysts either have to conform to the restrictions of the workflow language from the start, or they model their problems freely and transform the resulting specifications afterwards. A real issue here is that of suitability. In many cases the resulting workflows may be unnecessarily complex which impacts end-users who may wish to monitor the progress of their workflows. In this section two patterns are presented which illustrate typical restrictions imposed on workflow specifications and their consequences.

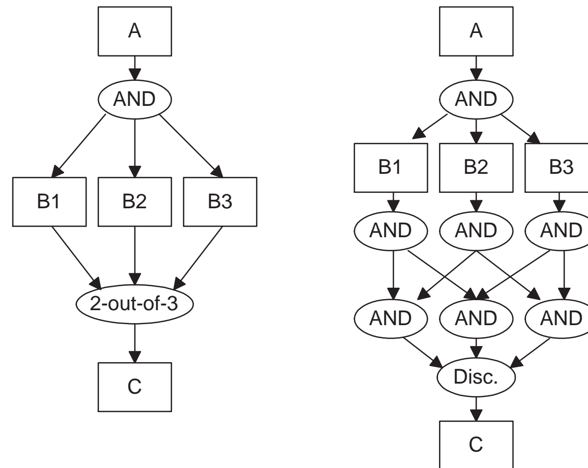

Figure 1: Implementation of 2-out-of-3-join using simple discriminator

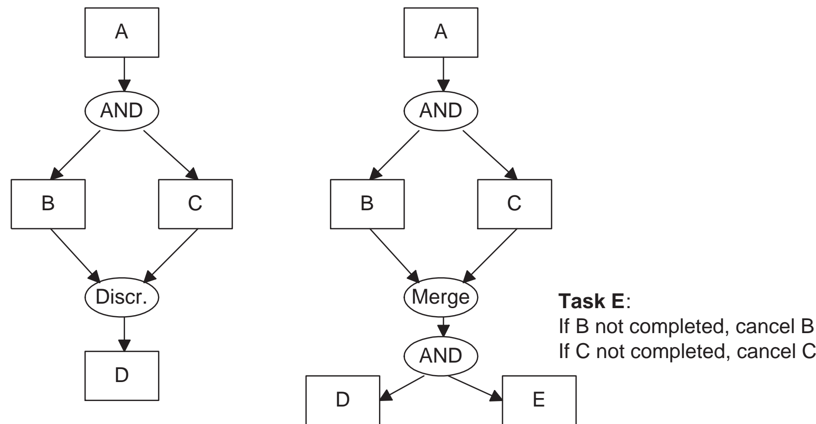

Figure 2: Design pattern for discriminator

Virtually every workflow engine has constructs that support the modelling 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).

<u>Pattern</u> **3 (Arbitrary Cycles)**
Description A point in a workflow process when one or more activities can be done repeatedly.
Synonyms Loop, iteration, cycle.
Examples

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

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

- Arbitrary cycles can typically be converted into structured cycles unless they contain one of the more advanced patterns such as multiple instances (see Pattern 5). The conversion is done either through auxiliary variables or through node repetition. A detailed analysis of such transformations and the extent to which they are possible, can be found in [KHB00]. Figure 3 provides an example of an arbitrary workflow converted to a structured workflow. Note that auxiliary variables Φ and Θ are required as we may not know which activities in the original workflow set the values of β and χ.

□

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

<u>Pattern</u> **4 (Implicit Termination)**
Description A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the workflow and no other activity can be made active (and at the same time the workflow is not in deadlock).
Examples

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

Problem Most workflow engines terminate the process when an explicit *Final* node is reached. Any current activities that happen to be running by that time will be aborted.
Solutions

- Some workflow engines (Staffware, MQSeries/Workflow, InConcert) would terminate the (sub)process when there is nothing else to be done

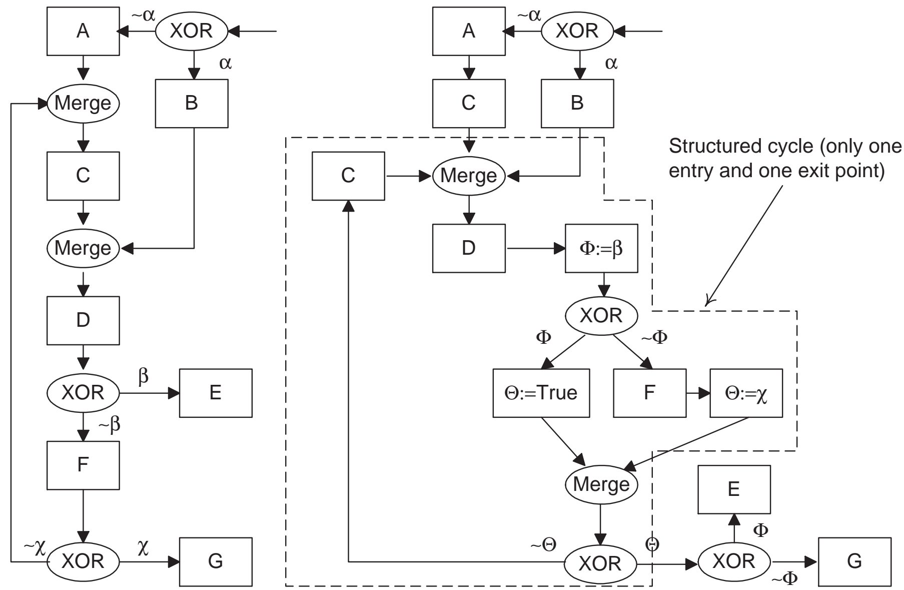

Figure 3: Implementation of arbitrary cycles

- 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. There are cases when it is not possible to do so. Clearly one of the cases when it is impossible is a model that involves multiple instances (see section 4). The required semantics is impossible to achieve without resorting to external triggers.

□

4 Patterns involving multiple instances of an activity

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

Pattern 5 (Multiple Instances With a Priori Runtime Knowledge)
Description For one case an activity is enabled multiple times. The number of instances of a given activity for a given case is variable and may depend on characteristics of the case or availability of resources [CCPP98, JB96]. but is known at some stage during runtime, before the instances of that activity have to be created.
Examples

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

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

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

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

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

- This pattern is a specific instance of Pattern 6 thus any implementation of Pattern 6 will be applicable.

- If there is a maximum number of possible instances, then the simple combination of XOR-split and AND-split constructs 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 one AND-split with the corresponding cardinality. The drawback of this solution is that the resulting workflow model can become large and complex and is bound by the maximum number of possible instances.

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

- As in many cases, the desired routing behaviour can be supported quite easily by making it more sequential. Simply use iteration to activate instances of the activity *sequentially*.

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

□

Pattern **6 (Multiple Instances With No a Priori Runtime Knowledge)**
Description For one case an activity is enabled multiple times. The number of instances of a given activity for a given case is not known during design time, nor it is known at any stage during runtime, before the instances of that activity have to be created.
Examples

- The requisition of a 100 computers results in a certain number of deliveries. The number of deliveries is unknown and varies for each case. Once each delivery is obtained, it can be determined whether a next delivery is to come by comparing the total number of delivered goods so far with the number of the goods requested.

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

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

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

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

- Similarly to Pattern 5, the desired routing behaviour can be supported quite easily by making it sequential.

□

Pattern **7 (Multiple Instances Requiring Synchronisation)**
Description For one case an activity is enabled multiple times. The number of instances may

not be known at design time. After completing all instances of that activity another activity has to be started.

Examples

- When booking a trip, the activity *book_flight* is executed multiple times if the trip involves multiple flights. Once all bookings are made, the invoice is to be sent to the client.
- The requisition of a 100 computers results in a certain number of deliveries. Once all deliveries are processed, the requisition has to be closed.

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

Solutions

- If the number of instances (or maximum number of instances) is known at design time, then it is easy to synchronise the multiple instances implemented through activity repetition by using basic synchronisation.
- If the language supports multiple instances *and* decomposition that does not terminate unless all activities are finished, then multiple instances can be synchronised by placing the workflow sub-flow containing the loop generating the multiple instances inside the decomposition block. The activity to be done once all instances are completed can then follow that block.
- MQSeries/Workflow's *Bundle* construct can be used when the number of instances is known at some point during runtime to synchronise all created instances.
- In most workflow languages none of these solutions can be easily implemented. The typical way to tackle this problem is to use external triggers. Once each instance of an activity is completed, the event should be sent. There should be another activity in the main process waiting for events. This activity will only complete after all events from each instance are received.

□

Figure 4 presents some design patterns for multiple instances. Workflow (a) can be implemented in languages supporting multiple concurrent instances of an activity as well as implicit termination (see Pattern 4). An activity B will be invoked here many times, activity C is used to determine if more instances of B are needed. Once all instances of B are completed, the subprocess will complete and activity E can be processed. Implicit termination of the subprocess is used as the synchronising mechanism for the multiple instances of activity B.

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

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

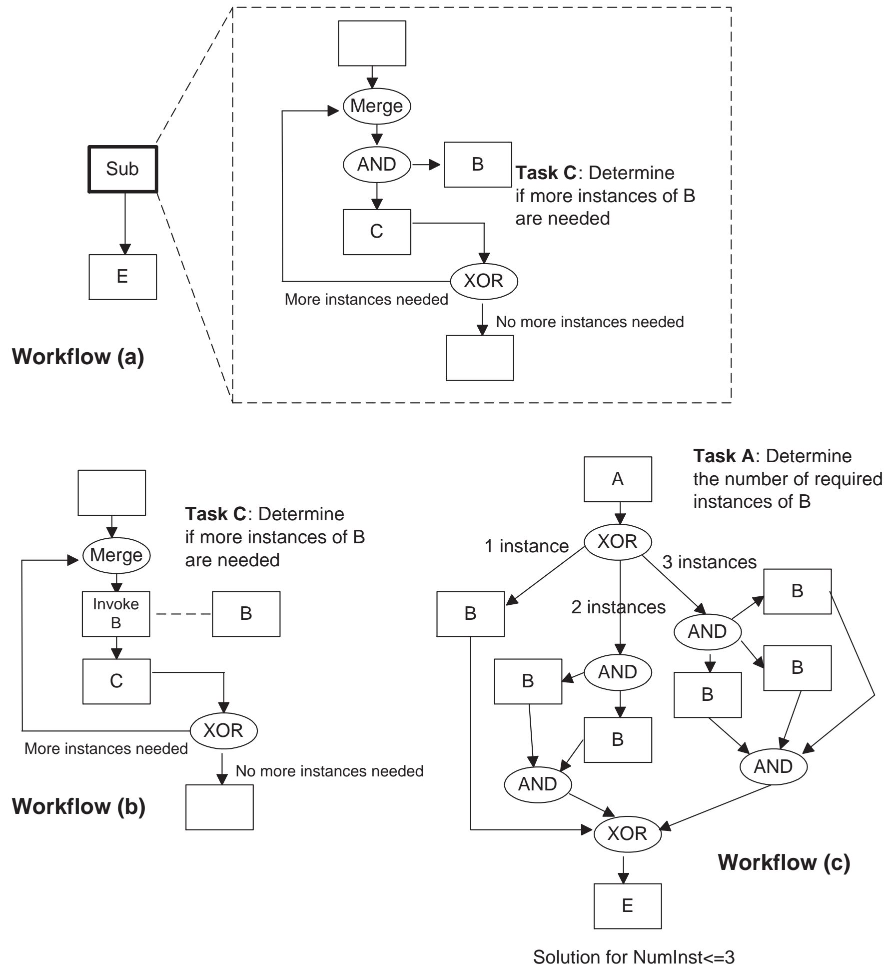

Figure 4: Design patterns for multiple instances

5 State-based Patterns

In real workflows, most workflow instances are in a state awaiting processing rather than being processed. Most computer scientists, however, seem to have a frame of mind, typically

derived from programming, where the notion of state is interpreted in a narrower fashion and is essentially reduced to the concept of data. As this section will illustrate, there are real differences between work processes and computing and there are business scenarios where an explicit notion of state is required. As the notation we have deployed so far is not suitable for capturing states explicitly, we adopt the variant of Petri-nets as described in [Aal98b] when illustrating the patterns in this section. Petri-nets provide *a* possible solution to modelling states explicitly (an example of a commercial workflow management system based on Petri-nets is COSA).

Moments of choice, such as e.g. 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 5 adopted from [Aal98b]. In this figure two workflows are depicted. In both workflows, the execution of activity A is followed by the execution of B or C. In workflow (a) the moment of choice is as late as possible. After the execution of activity A there is a "race" between activities B and C. If the external message required for activity C (this explains the envelope notation) arrives before someone starts executing activity B (the arrow above activity B indicates it requires human intervention), then C is executed, otherwise B. In workflow (b) the choice for either B or C is fixed after the execution of activity A. If activity B is selected, then the arrival of an external message has no impact. If activity C is selected, then activity B cannot be used to bypass activity C. Hence, it is important to realize that in workflow (a), both activities B and C were, at some stage, simultaneously scheduled. Once an actual choice for one of them was made, the other was disabled. In workflow (b), activities B and C were at no stage scheduled together.

Example 5.1 As a concrete example of the difference between explicit and implicit choices, imagine activity A of Figure 5 to represent the activity *send_questionnaire*, and activities B and C, the activities *time_out* and *process_questionnaire*. The activity *time_out* requires a time trigger, while the activity *process_questionnaire* is only to be executed if the complainant returns the form that was sent (hence an external trigger is required for its execution). Clearly, the moment of choice between *process_questionnaire* and *time_out* should be as late as possible. If this choice was modelled as an implicit XOR-split, 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. □

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

Pattern 8 (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)

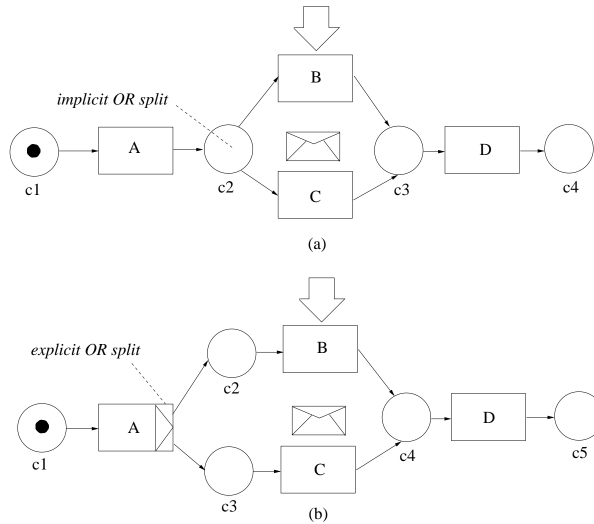

Figure 5: Illustrating the difference between implicit (a) and explicit (b) XOR-splits

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.

Examples

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

- See the choice between B and C in Figure 5.

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

Solutions

- Assume that the workflow language being used supports AND-splits and the cancellation of activities. The implicit XOR-split can be realized by enabling all alternatives via an AND-split. Once the processing of one of the alternatives is started, all other alternatives are cancelled. Consider the implicit choice between B and C in Figure 5(a). After A both B and C are enabled. Once B is selected/executed, activity C is cancelled. Once C is selected/executed, activity B is cancelled. Note that the solution does not always work because B and C can be selected/executed concurrently.

- Another solution to the problem is to replace the implicit XOR-split by an explicit XOR-split, i.e., an additional activity is added. All triggers activating the alternative branches are redirected to the added activity. Assuming that the activity can distinguish between triggers, it can activate the proper branch. Consider the example shown in Figure 5. By introducing a new activity E after A and redirecting triggers from B and C to A, the implicit XOR-split can be replaced by an explicit XOR-split based on the origin of the first trigger. Note that this solution moves part of the routing to the application or task level.

□

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

Pattern 9 (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.
Solutions

- A very simple, but unsatisfactory solution, is to fix the order of execution, i.e., instead of using parallel routing, sequential routing is used. Since the activities can be executed in an arbitrary order, a solution using a predefined fix order may be acceptable. However, by fixing the order, flexibility is reduced and the resources cannot be utilized to their full potential.

- Another solution is to predefine a number of alternative sequences and before execution one sequence is selected using a XOR-split. A drawback is that the order is fixed before

the execution starts. Moreover, the workflow model may become quite complex and large by enumerating all possible sequences.

- By using the deferred XOR-split the order does not need to be fixed before the execution starts, i.e., the deferred XOR-split allows for on-the-fly selection of the order. Unfortunately, the resulting model typically has a "spaghetti-like" structure.

- For workflow models based on Petri nets, the interleaving of activities can be enforced by adding a place which is both an input and output place of all potentially concurrent activities. The AND-split adds a token to this place and the AND-join removes the token. It is easy to see that such a place realizes the required "mutual exclusion". See Figure 6 for a example where this construct is applied. Note that, unlike the other solutions, the structure of the model is not compromised.

□

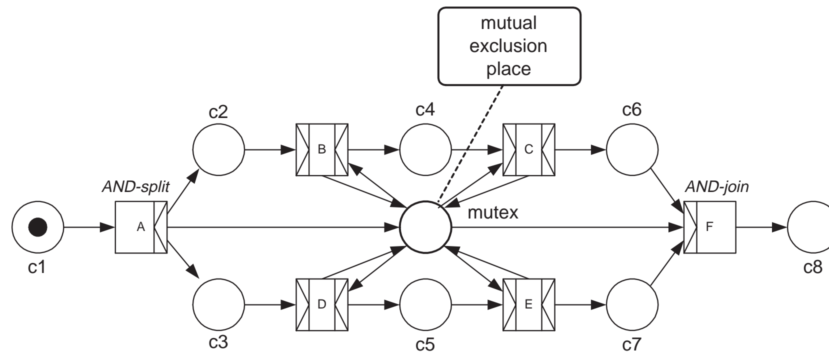

Figure 6: The execution of B, C, D, and E is interleaved by adding a mutual-exclusion place

Noe that the solution shown in Figure 6 is only possible because mutual exclusion can be enforced by place *mutex* (i.e., state information shared among the activities). The next pattern, Pattern 10, allows for testing whether a case has reached a certain phase. By explicitly modelling 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 5.2 Consider the workflow process for handling complaints (see Figure 7). First the complaint is registered (activity *register*), then in parallel a questionnaire is sent to the complainant (activity *send_questionnaire*) and the complaint is evaluated (activity *evaluate*). If the complainant returns the questionnaire within two weeks, the activity *process_questionnaire* is executed. If the questionnaire is not returned within two weeks, the result of the questionnaire is discarded (activity *time_out*). Based on the result of the evaluation, the complaint is processed or not. The actual processing of the complaint (activity *process_complaint*) is delayed until the questionnaire is processed or a time-out

has occurred. The processing of the complaint is checked via activity *check_processing*. Finally, activity *archive* is executed. □

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

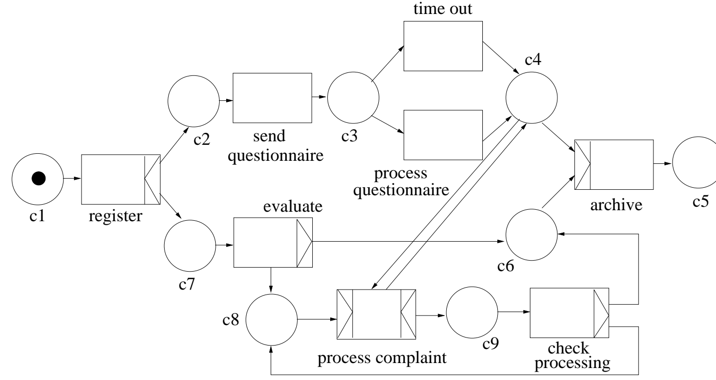

Figure 7: The state in-between the processing/time-out of the questionnaire and archiving the complaint (i.e. place *c4*) is an example of a milestone

<u>Pattern</u> 10 (Milestone)
Description The enabling of an activity depends on the case being in a specified state, i.e., the activity is only enabled if a certain milestone has been reached which did not expire yet. Consider three activities A, B, and C. Activity A is only enabled if activity B has been executed and C has not been executed yet, i.e., A is not enabled before the execution B and A is not enabled after the execution C.
Synonyms Test arc, deadline (cf. [JB96]), state condition.
Examples

- In a travel agency, flights, rental cars, and hotels may be booked as long as the invoice is not printed.
- A customer can withdraw purchase orders until two days before the planned delivery.
- A customer can claim airmiles until six months after the flight.
- The construct involving activity *process_complaint* and *c4* shown in Figure 7.

Problem The problem is similar to the problem mentioned in Pattern 8: There is a race between a number of activities and the execution of some activities may disable others. Note that in Figure 7 activity *process_complaint* may be executed an arbitrary number of times, i.e., it is possible to bypass *process_complaint*, but it is also possible to execute *process_complaint*

several times.

Solutions

- Consider three activities A, B, and C. Activity A can be executed an arbitrary number of times before the execution of C and after the execution of B. Such a milestone can be realized using Pattern 8. After executing B there is an implicit XOR-split with two possible subsequent activities: B and C. If B is executed, then the same implicit XOR-split is activated again. If C is executed, B is disabled by the implicit XOR-split construct. Note that this solution only works if the execution of B is not restricted by other parallel threads. For example, the construct cannot be used to deal with the situation modelled in Figure 7 because *process_complaint* can only be executed directly after a positive evaluation or a negative check, i.e., the execution of *process_complaint* is restricted by both parallel threads.

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

□

Having introduced the milestone pattern another solution to Pattern 7 can be given (see Figure 8). This solution uses a mixture of a loop construct to enable multiple instances of B in parallel, and a milestone and XOR-split to detect the completion of all instances. Note that the enabling is done sequentially, the actual execution, however, is done in parallel. The AND-split X enables one instance of B and activates Y. The XOR-split Y checks whether all instances have been enabled. The AND-join/OR-split Z uses a milestone-like construct and is activated for every instantiation of B. Z determines whether all instances have been processed, i.e., it waits for the next completion of an instance of B or enables C.

It is interesting to think about the reason why many workflow products have problems dealing with patterns 8, 9, and 10. The systems that abstract from states are typically based on messaging, i.e., if an activity completes, it notifies or triggers other activities. This means that activities are *enabled* by the receipt of one or more messages. Patterns 8, 9, and 10 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

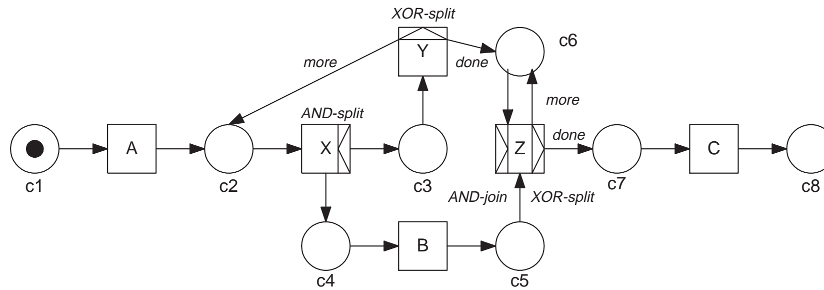

Figure 8: Multiple activation of activity B using the milestone pattern

patterns 8, 9, and 10 further complicates the use of asynchronous communication mechanisms such as message passing using "negative messages" (e.g., messages to cancel activities).

6 Epilogue

This paper presented a number of advanced 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.

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.

[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.

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

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

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

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

[KHB00] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On Structured Workflow Modelling. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE'00)*, Stockholm, Sweden, June 2000. (to appear).

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

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

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

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

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

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

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

A Products

This appendix introduces the products mentioned in this paper and supplies some background information.

COSA [SL96] is a Petri-net-based workflow management system developed by Ley GmbH (formerly operating under the names Software Ley and COSA Solutions). Ley GmbH is a German company based in Pullheim (Germany) and is part of the Baan Consortium (i.e., a member of the Vanenburg Group). COSA is one of the leading workflow management systems in Europe and can be used as a stand-alone workflow system or as the workflow module of the Baan IV ERP system. COSA will also be used as the workflow engine of the new BaanSeries platform. For our evaluation we used version 2.0. The modelling language of COSA consists of two types of building blocks, i.e., activities (tasks or transitions) and conditions (places), and corresponds to a variant of Petri nets extended with control data.

Visual WorkFlo is one of the market leaders in the workflow industry. It is part of the FileNet's Panagon suite that includes also document management and imaging servers. Visual WorkFlo is one of the oldest and best established products on the market. Since its introduction in 1994 it managed to gain a respectable share of all worldwide workflow applications. FileNet as a corporation ranks amongst the top 60 software companies in the world (Software magazine) - with offices in 13 countries and over 650 Value Added Resellers building solutions on top of Panagon's suite. The workflow modelling language of Visual WorkFlo is highly structural and is a collection of tasks and routing elements such as Branch (XOR-SPLIT), While (structured loop), Static Split (AND-SPLIT), Rendevous (AND-JOIN), Release.

Forté Conductor is a workflow engine that is an add-on to Forté's powerful development environment, Forté 4GL (formerly Forté Application Environment). Forté Software has recently (in October 1999) been acquired by Sun Microsystems. Conductor's engine is based on experimental work performed at Digital Research and its modelling language is powerful and flexible. The workflow model in Conductor comprises a set of tasks connected with transitions (called *Routers*). Each transition has associated transition conditions. Each activity has a trigger that determines the semantics of that activity if it has more than one incoming transition. The triggers are flexible enough for easy specification of OR-JOIN, AND-JOIN and any type of N-out-of-M join (see pattern 2).

MQSeries/Workflow is the successor of IBM's major workflow offering, FlowMark. FlowMark was one of the first workflow products that was independent from document management and imaging services. It has been renamed to MQSeries/Workflow after a major move from the proprietary middleware to middleware based on MQSeries product. The workflow model consists of activities linked by transitions. Other than a decomposition block, there are no other special modelling constructs available. The workflow engine of MQSeries/Workflow has a unique execution semantics in that it propagates a *False Token* for every transition with a condition evaluating to False. This allows for every activity that has more than one incoming transition to act as a synchronising OR-JOIN (see pattern 1).

Staffware [Sta97] is one of the leading workflow management systems. Staffware is authored and distributed by Staffware PLC. Staffware PLC is headquartered in Maidenhead (UK), operates through offices in 15 countries and a network of 360 partners, resellers and OEMs. We did not use the most recent version of Staffware (i.e., Staffware 2000) which was released in the last quarter of 1999. Instead, we used Staffware 97 [Sta97] for our evaluation. This version of Staffware is used by more than 550,000 users worldwide and runs on more than 4500 servers. In 1998, it was estimated by the Gartner Group that Staffware has 25 percent of the global market [Cas98]. The routing elements used by Staffware are the Start, Step, Wait, Condition, and Stop. The Step corresponds to a task 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).

Verve is a relative newcomer to the workflow market. What makes it an interesting workflow product is that it has been designed from the ground up as an embeddable workflow engine. The workflow engine of Verve is very powerful and amongst other features allows for multiple instances and dynamic modification of running instances. Verve workflow model consists of activities connected by transitions. Each transition has an associated transition condition. Extra routing constructs such as synchroniser and discriminator are supported.

I-Flow is a workflow offering from Fujitsu that can be seen as a successor of the well-established workflow engine from the same company, TeamWare. I-Flow is web-centric and has a Java/CORBA based engine built specifically for Independent Software Vendors and System Integrators. The workflow model in I-Flow consists of activities and a set of routing constructs connected by transitions (called *Arrows*). Routing constructs include Conditional Node (XOR-SPLIT), OR-NODE (Merge), AND-NODE (synchroniser). AND-SPLIT can be modelled implicitly by providing an activity with more than one outgoing transition. Decomposition is supported as well as asynchronous subprocess invocation. There is no support for multiple instances.

InConcert has been established in 1996 as a Xerox fully-owned subsidiary. In 1999 it has been bought by TIBCO Software. InConcert 2000 is the newest version of their flagship workflow offering. A InConcert workflow definition is called a "job". A job can contain none, one or many tasks. A task is either simple or compound. A task can be connected to an arbitrary number of other tasks but circular dependencies are not allowed. Each task has a perform condition attached to it. The default setting of the perform condition is "true" such that tasks can be executed in general. If the perform condition evaluates to "false" the task is skipped. If a task is skipped then the subsequent tasks are not skipped automatically. Conditional branching or case branching can be achieved by parallel tasks with different perform conditions.

SAP R/3 Workflow SAP is the main player in the market of ERP systems. Its R/3 software suite includes an integrated workflow component that we have evaluated independently of the rest of R/3. SAP workflow models are designed using so-called Event-controlled Process Chains (EPS). The control-flow perspective of EPS consists of set of functions (tasks), events and connectors (AND, XOR, OR).

Changengine is a workflow offering from HP, the second largest computer supplier in the world. Version 3.0 of the product has been introduced in 1998 and it is focused on high performance and support for dynamic modifications. Workflow models in Changengine consist of a set of work nodes and routers linked by arcs. A work node can have only one incoming and one outgoing arc. If more transitions are required, they have to be created explicitly through the router node. Modelling of arbitrary loops is allowed.

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