Advanced Workflow Patterns

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

[1] Eindhoven University of Technology, Faculty of Technology and Management, Department of Information and Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands. w.m.p.v.d.aalst@tm.tue.nl

[2] Distributed Systems Technology Centre, The University of Queensland, Brisbane Qld 4072, Australia, abarros@dstc.edu.au

[3] Cooperative Information Systems Research Centre, Queensland University of Technology, P.O. Box 2434, Brisbane Qld 4001, Australia, arthur@icis.qut.edu.au

[4] Mincom Limited, P.O. Box 1397, Brisbane Qld 4001, Australia, bartek@mincom.com

Abstract. Conventional workflow functionality like task sequencing, split parallelism, join synchronization and iteration have proven effective for business process automation and have widespread support in current workflow products. However, 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 modeling 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. These patterns provide the basis for an in-depth comparison of 12 workflow management systems. As such, the work reported in 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.

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

* 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.

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. [1, 4–6, 9, 10, 12–14, 16, 17]).

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

Clearly, the control flow perspective provides an essential insight into a workflow specification's effectiveness. The data flow perspective rests on it, while the organizational and operational perspectives are ancillary. If workflow specifications are to be extended to meet newer processing requirements, control flow constructors require a fundamental insight and analysis. Currently, most workflow languages support the basic constructs of sequence, iteration, splits (AND and OR) and joins (AND and OR) - see [13]. 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 [15], a pattern "is the abstraction from concrete form which keeps recurring in specific non-arbitrary contexts". Gamma et al. [8] 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. [7]).

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 [8], 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 only four workflow patterns. These patterns are just a small sample of the many patterns we have identified. In [3] we report 26 patterns. 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. The goal of this paper is not to provide a comprehensive overview of workflow functionality: It only shows the flavor of the research that has been conducted. For a more complete overview we refer to [3].

2 Advanced Synchronization Patterns

In most workflow engines two basic forms of synchronization 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 synchronize 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 synchronization patterns. An example of such an advanced synchronization pattern is the so-called Synchronizing Merge.

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

- 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 synchronize and when to merge. Synchronizing alternative flows leads to potential deadlocks and merging parallel flows may lead to unwanted, multiple execution of the activity that follows the standard OR-join construct.
Solutions

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

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

□

The synchronizing merge is just an example of an advanced synchronization pattern. In [3] we have identified additional ones such as the Multi-merge, the Discriminator, and the N-out-of-M Join.

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.

An example of a typical structural requirement imposed by some of the 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 one is reached. Again, most business models do not follow this pattern - it is more natural to think of a business process as terminated once there is nothing else to be done.

Pattern 2 (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.
- 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.

□

Another pattern described in [3] is the so-called Arbitrary Cycle (cf. [11]). 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).

4 Patterns involving multiple instances of an activity

Many workflow management systems have problems with the 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. As an example we discuss one pattern dealing with multiple instances.

Pattern 3 (Multiple Instances Requiring Synchronization)
Description For one case an activity is enabled multiple times. The number of instances may not be known at design time. After completing all instances of

that activity another activity has to be started.
Examples

- When booking a trip, the activity *book_flight* is executed multiple times if the trip involves multiple flights. Once all bookings are made, the invoice is to be sent to the client.
- The requisition of 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 and Verve) do not provide any construct that would allow for synchronization of these instances. Languages that support the asynchronous subprocess invocation (e.g. Visual WorkFlo through the *Release* construct) do not provide any means for for the synchronization of spawned off subprocesses.
Solutions

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

□

Pattern 3 is just an example of a pattern dealing with multiple instances. In [3] we have identified additional ones. Figure 1 illustrates some design patterns for dealing with multiple instances. Workflow (a) can be implemented in languages supporting multiple concurrent instances of an activity as well as implicit termination (see Pattern 2). An activity B will be invoked here many times, activity C is used to determine if more instances of B are needed. Once all instances of B are completed, the subprocess will complete and activity E can be processed. Implicit termination of the subprocess is used as the synchronizing mechanism for the multiple instances of activity B. Workflow (b) can be implemented in languages that do not support multiple concurrent instances. Activity B is invoked asynchronously, typically through an API. There is no easy way to synchronize

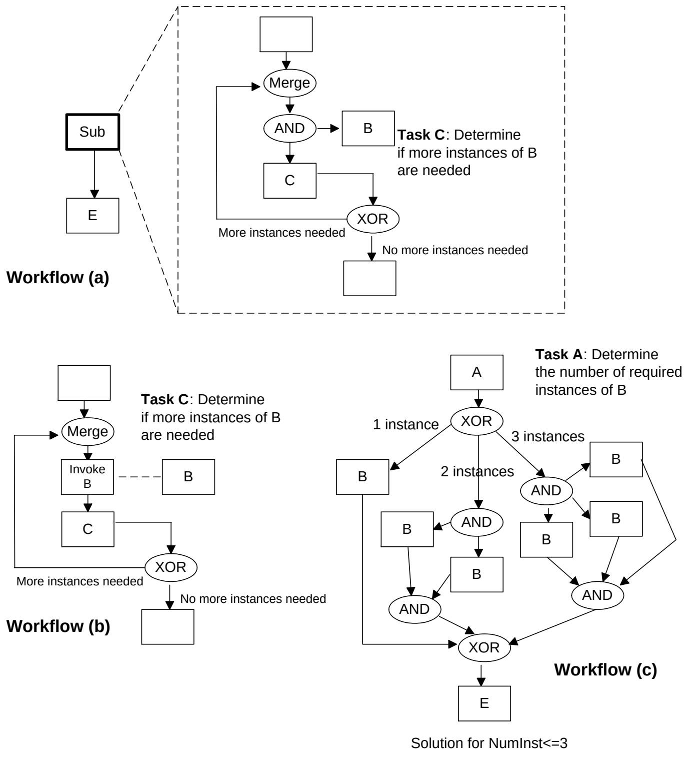

Fig. 1. Design patterns for multiple instances

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.

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 [2] when illustrating the patterns in this section. Petri-nets provide *a* possible solution to modeling 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 2 adopted from [2]. 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.

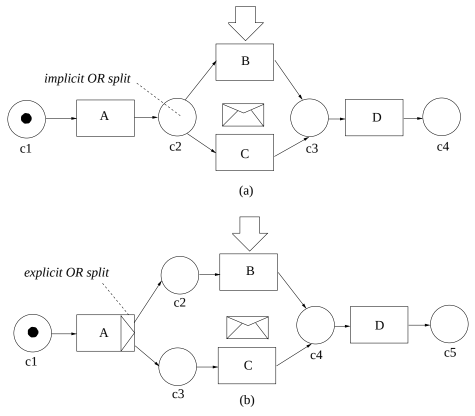

Fig. 2. Illustrating the difference between implicit (a) and explicit (b) XOR-splits

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

Pattern 4 (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.
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 2. Activity A may represent the sending of a form to a customer. Activity C corresponds to the processing of the form once it is returned. Activity B corresponds to situation where the form is not received in time and some alternative action is taken.

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 2(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 2. 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.

$\square$

In [3] we have identified several patterns related to the Deferred Choice. An example of such a pattern is the so-called Milestone. In this pattern one branch of a parallel process is offered a time window by another branch to executed certain parts of the process. Other related patterns are Cancel Activity, Cancel Case, and Interleaved Parallel Routing. These patterns have in common that an explicit notion of states is required and that they are supported by only a few workflow management systems.

It is interesting to think about the reason why many workflow products have problems dealing with state-based patterns. 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. 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 patterns such as the deferred choice (i.e., Pattern 4) further complicates the use of asynchronous communication mechanisms such as message passing using "negative messages" (e.g., messages to cancel activities).

6 Epilogue

The four workflow patterns described in this paper correspond to routing constructs encountered when modeling and analyzing workflows. These patterns illustrate the more complete set of 26 workflow patterns reported [3]. 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.

We have used a comprehensive set of workflow patterns to compare the functionality of 12 workflow management systems (COSA, Visual Workflow, Forté Conductor, Meteor, Mobile, MQSeries/Workflow, Staffware, Verve Workflow, I-Flow, InConcert, Changengine, and SAP R/3 Workflow), cf. [3]. From the comparison it is clear that no tools support all the selected patterns. In fact, many of these tools only support a fraction of these patterns and the best of them only support about 50%. Specifically the limited support for state-based patterns and advanced synchronization patterns (e.g., multiple instances, merge, N-out-of-M) is worth noting. 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. Therefore, it is worthwhile to use the set of patterns given in [3] as a check list when selecting a workflow product.

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

References

1. 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.
2. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
3. W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, and B. Kiepuszewski. Workflow Patterns. Unpublished (46 pages), 2000.
4. W.M.P. van der Aalst and A.H.M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based Approach. *Information Systems*, 25(1):43–69, 2000.
5. 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-Verlag, Berlin, 1998.
6. 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.
7. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, Massachusetts, 1997.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
9. 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.
10. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, 1996.
11. B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On Structured Workflow Modelling. In B. Wangler and L. Bergman, editors, *12th International Conference, CAiSE 2000*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445, Stockholm, Sweden, June 2000. Springer-Verlag, Berlin.
12. T.M. Koulopoulos. *The Workflow Imperative*. Van Nostrand Reinhold, New York, 1995.
13. P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997.
14. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
15. D. Riehle and H. Züllighoven. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.
16. T. Schäl. *Workflow Management for Process Organisations*, volume 1096 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.
17. WFMC. Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition, Brussels, 1996.