Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows

Bartosz Kiepuszewski, M.Sc.

A dissertation presented to the
Faculty of Information Technology
Queensland University of Technology
in fulfilment of the requirements for the degree of
Doctor of Philosophy

November 13, 2002

Abstract

Workflow management has been a research area that attracted significant interest in the last decade. In spite of this, little consensus has been reached as to what the essential ingredients of workflow specification languages should be. Consequently many workflow management systems have been developed that are based on different paradigms and include different feature sets. These differences result in various levels of *suitability* and *expressive power*.

The challenge, which we undertake in this thesis, is to provide a comprehensive analysis of control flow aspects of workflow specifications. We start by analysing modelling languages of eight commercially available workflow systems.

Next we identify the requirements for workflow control flow through the use of workflow *patterns*. Ranging from very simple to more complex, we identify the business requirements that need to be addressed by any workflow management system that is to be used to support a wide range of business processes. Through matching these patterns with the modelling capabilities of the workflow systems that we have under review, we provide a thorough analysis of these systems and identify shortcomings in their modelling language approaches.

Finally, we establish a theoretical foundation that is subsequently used to provide a formal analysis of the expressive power and suitability of different approaches to modelling control flow in workflows. We identify four major evaluation strategies and assess their theoretical expressive power. Additionally we analyse some of the more advanced concepts related to control flow modelling for workflows such as termination, deadlock, decomposition and use of data flow to augment control flow specification. The results obtained as part of this work should not only aid those developing workflow specifications in practice, but also those developing new workflow engines.

Contents

List of Figures

List of Tables

Publications based on this thesis

- A.H.M. ter Hofstede and **B. Kiepuszewski**. Formal Analysis of Deadlock Behaviour in Workflows. Technical report FIT-TR-1999-03, Queensland University of Technology/Mincom, Brisbane, Australia, April 1999.

- **B. Kiepuszewski** and A.H.M. ter Hofstede. Experiences with Embedding a Workflow Engine in Mims. In *Proceedings of the 3rd Workflow Management Conference*, Muenster, Germany, November 1999.

- **B. Kiepuszewski**, A.H.M. ter Hofstede, and C. Bussler. On Structured Workflow Modelling. In B. Wangler and L. Bergman, editors, *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE'00)*, volume 1789 of *Lecture Notes in Computer Science*, pages 431-445, Stockholm, Sweden, June 2000. Springer-Verlag.

- W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, and **B. Kiepuszewski**. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *Fifth IFCIS International Conference on Cooperative Information Systems (CoopIS'2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18-29, Eilat, Israel, September 2000. Springer-Verlag.

- W.M.P. van der Aalst, A.H.M. ter Hofstede, **B. Kiepuszewski** and A.P. Barros. Worklfow Patterns. Technical Report WP 47, BETA Research Institute, Einhoven University of Technology, Eindhoven, The Netherlands, August 2000. Submitted for publication in *Distributed and Parallel Databases*.

- **B. Kiepuszewski**, A.H.M. ter Hofstede and W.M.P. van der Aalst. Fundamentals of Control Flow in Workflows. Technical report, FIT-TR-2001-01, Queensland University of Technology/Mincom, Brisbane, Australia, January 2001. Conditionally accepted for publication in *Acta Informatica*.

Declaration

I declare that to the best of my knowledge and belief that the work presented in this thesis is my own work, except as otherwise acknowledged in the text. The material has not been submitted, either in whole or in part, for a degree at this or any other university.

Bartek Kiepuszewski

Brisbane, Australia

20 March, 2002

Acknowledgements

I am deeply indebted to my supervisor, A/Professor Arthur ter Hofstede, for his guidance and patience during the course of my doctoral studies at the Queensland University of Technology. Arthur's work ethic, commitment and inspirational manner has had a role in the successful completion of this thesis.

I would like to thank the management team at both Mincom Limited, my former employer, and Infovide, my current employer, for providing the opportunity to me to undertake my doctoral studies in conjunction with my work responsibilities. Special thanks go especially to David Cox, John Benders, William Ferguson, and many other Mincom staff members as well as Kuba Moszczyński, Borys Stokalski and other Infovide staff members who directly or indirectly made this work possible.

I would also like to offer my deepest gratitude towards many colleagues from different organizations who provided me with a great deal of additional insight during many hours of collaborative work. Special thanks should especially go to Wasim Sadiq and Alistair Barros of DSTC, Wil van der Aalst of Eindhoven University of Technology and Christoph Bussler of Oracle.

Last but not least I would like to thank my wife, Gosia for her continuous support, love and patience as well as my little boy Adam for his understanding and presence during darkest moments of doubt.

Chapter 1

Introduction

Every modern business environment is characterised by an extensive set of business processes that needs to be followed to achieve stated business objectives. In the past work was passed from one participant (or worker) to another manually. As the work was delivered to people, each participant could assume that work was ready for processing. The focus of information technology was on automating individual tasks performed by participants so that they can be completed in a more timely and efficient manner.

In recent years, the possibility of automating the coordination of the processes themselves has been explored, resulting in an area of research and technology commonly referred to as *workflow technology*. In one of the defining research papers on workflow technology ([GHS95]) Georgakopoulos et al. associate workflow technology with facilitating business process specification, reengineering and automation while defining workflow as a "collection of tasks organized to accomplish some business process". The Workflow Management Coalition, the industry standardisation group comprising many leading workflow vendors defines workflow as "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules" ([Wor99b]).

The traditional areas of business process modelling, business process coordination and document and image management along with emergent areas such as business-to-business and business-to-consumer interactions propelled a great diversity of different workflow management systems (WfMS) to be available either as commercial products or research prototypes. Despite ongoing standardisation, these products are often based on different paradigm and they support a large variety of process modelling languages (see e.g. [Aal98a, Aal98c, Ell79, EN93, JB96, Kou95, LR99, Law97, Sch96, DKTS98]).

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, choice, parallelism, and 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 flow 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 language's effectiveness. The data flow perspective rests on it, while the organizational and operational perspectives are ancillary.

1.1 Problem Statement

It is perhaps surprising that in discussions on workflow products, emphasis is hardly ever on the workflow languages used, rather focus is almost exclusively on operational and architectural aspects. Workflow modelling requirements, even though virtually ignored at the early stage of many projects, always turn out to play an important role as they heavily impact on the quality of business process analysis and ultimately the final workflow design.

Workflow modelling requirements naturally derive from generic requirements for information modelling techniques as promulgated by e.g. [Hof93] which are presented as a set of principles that should characterize any good modelling language. In this Thesis the main focus in on three such principles: suitability, expresiveness and formality.

The thought that a "silver bullet" exists for the problem of information systems development has dogged research for a long time. These days however it is recognised that there isn't a "one best way" approach to information systems development (see e.g. [AW91]). The choice of technique very much depends on the problem domain. This implies that modelling techniques should be *suitable* for the domain that they are to be used in. Suitability problems in workflow languages typically manifest themselves in different ways to solving the same modelling problem, some being direct,

"natural" solutions, others requiring elaborate workarounds. Suitability is a subjective notion - we will often argue for some approaches to be more suitable than others, but it is possible that some readers may disagree.

The 100% principle ([Gri82]) states that a conceptual model should describe all relevant static and dynamic aspects of the problem at hand. This implies that a language should have sufficient *expressive power*. In contrast to suitability, expressiveness is an objective criterion. Given a certain set of modelling constructs, it is often possible to show that certain processes can, or cannot be modelled.

There are many issues related to the expressive power and suitability of workflow languages. Some languages allow multiple instances of the same activity 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. It is often unclear how these different approaches to workflow modelling affect the expressive power and suitability of a given modelling language.

Finally, *formality* is a principle, that traditionally in the field of information systems, has not attracted a lot of attention. Many languages have been proposed without a *formal* foundation and it has increasingly become clear that this is not a desirable situation. The use of informal languages, i.e. languages without a formal semantics, easily leads to specifications that are inherently ambiguous. In addition to that, such specifications cannot be formally validated, verified, or analysed with respect to their expressive power. We believe that workflow specifications in particular need a good formal foundation given that workflow models are rare examples of graphical models that not only serve as a communication mechanism between users and analysts, or analysts and designers, but they provide a specification of a system (a workflow process) that is directly executable by a workflow engine.

Given the existence of many different modelling techniques, how can we asses their respective suitability and expressive power? Given a business process that needs to be automated, should the choice of a workflow system be made based on the modelling capabilities of this system or are the architectural and operational requirements the only ones worth consideration? Is it possible to interchange process models from two different workflow languages given that they have fundamentally different semantics? Are all workflow languages essentially the same and is the difference mainly terminology and graphical notation or are they based on fundamentally different paradigms making interoperability a difficult proposition? There seems to be no common theory for workflow specification similar to relational model theory for relational database management systems hinting at the possibility that the answer to the above questions is not an easy one. These are the questions we would like to answer in this Thesis.

1.2 Approach

We will provide answers to the questions raised in the previous section through a comprehensive analysis of the control flow perspective of workflows with particular focus on expressiveness and suitability of different workflow modelling languages used in practice. We will provide both a practical approach for evaluating a given modelling language as well as a theoretical framework for establishing the limits of expressive power in a precise, formal manner. This should not only help analysts specifying workflows in practice, as they may have to understand fundamental limits of the workflow language they use or have to map their specifications to, but also developers designing new workflow engines, as the results presented may prevent them from imposing restrictive constraints on workflow specifications, or may, in some cases, provide them with certain useful equivalence preserving transformations.

In the first part of this Thesis 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]).

In the second part of this Thesis we provide a framework necessary to establish the *theoretical* limit of a given modelling technique. Our experience in formal methods is that providing a formal specification not only enables reasoning about the properties of a workflow specification (such as complexity of verification, expressiveness, etc) but also greatly contributes to a thorough understanding of the semantics of the constructs in question. An interesting experience was gained with Verve Workflow, which supports a concept referred to as *discriminator*. The discriminator allows multiple parallel execution paths to be synchronised in such a way that only the first path to finish initiates the rest of the execution. Naturally, this concept requires care when used in a loop. The formal semantics assigned by us to the discriminator, which we present in Section 6.3, was distilled from the documentation and would lead to proper behaviour when used in a loop. Unfortunately, it turned out that the discriminator behaved differently when used in an actual workflow. This has led to a recommendation to implement this concept as specified by the formal Petri net and to the best of our knowledge the new version of Verve Workflow is going to have the semantics of the discriminator altered as per our suggestions.

The other reason to specify formal semantics for workflow specification languages is to avoid ambiguity. Notions which have similar names and have similar syntactical

restrictions may have an entirely different semantics. As we will show throughout this thesis, even the most fundamental workflow modelling concepts such as AND-Joins and OR-Joins, given the ambiguous definition of their semantics by the Workflow Management Coalition (see Section 2.1), do not have an uniform implementation, hence making workflow interoperability difficult if not impossible.

Based on the formal foundation the latter part of the Thesis is devoted to establishing a number of properties of different workflow modelling languages with the aim of providing a strong, theoretical argumentation for the choice of one approach over another.

1.3 Related Work

Most earlier works on workflow systems emphasize the separation of workflow description and execution levels ([BW95]), build time and run time ([JB96]), modelling and enactment ([GHS95]). Various workflow modelling approaches have been proposed. Among them is the MOBILE process model presented in [Jab94] by Jablonski. His approach recognizes the need to combine methods from organizational theory, behavioural theory, decision theory and database theory, in order to provide a complete specification for a workflow process. In [CCPP95] Casati et al propose a WFDL (WorkFlow Description Language) where a workflow model comprises workflow tasks and routing tasks. The latter concept can carry different semantics depending on the type of the routing task, of which particularly interesting are partial and iterative joins. In [RD97] Reichert and Dadam present fundamentals of the ADEPT Workflow Model. This model is based on the concept of symmetrical control structures, where various structures such as splits, joins and loops are specified as symmetrical blocks with explicit start and end points. Finally in [KG99] Kradolfer and Geppert present a TRAMs workflow model in which control flow is specified through start and end conditions of workflow tasks.

None of this work, however, focuses on comparative study of different approaches towards workflow modelling. Typically, new modelling techniques are proposed without sufficient discussion of comparative strengths and weaknesses with existing techniques. The comparative studies published by prestigious consulting companies such as The Butler Group ([Mak96]) or e.g. Accenture, Andersen, Ernst & Young, Deloitte & Touche and Price Waterhouse Coopers, 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?).

Few new techniques proposed are accompanied by a formal specification of the semantics. Interestingly, it is sometimes the case that other researchers "second-guess"

the intentions of the original authors and attempt to provide such a formal specification (see e.g. [Aal98b]). To some extent this is a challenge we are faced with in this Thesis. Among techniques that are used to provide a formal foundation for workflow specification, the most popular is through translation to, or direct application of Petri nets or some variant thereof ([AAH98, EN93, AHH94, AMP94, HHSW96]). In [Aal98a] van der Aalst argues that using Petri nets as the underlying formal foundation for workflows provides a workflow modeller with several theoretically proven analysis techniques that already have been used for Petri nets thus allowing for formal verification of workflow models. Additionally he hints on the expressiveness issue pointing out that, in contrast to activity-based workflow models, Petri nets provide means of distinguishing states of the model (and constituent tasks), so that task states such as initiation, activation, and termination, can be clearly identified.

Another attempt at providing a theoretical foundation for workflow specification can be found in [HN93] where process algebra ([BK84]) was used to provide the semantics of task structure diagrams and bisimulation is used as the equivalence notion for establishing equivalence between different workflow models. In [SO99] Sadiq and Orlowska represent a workflow process as a direct, acyclic graph and defines its semantics using a set of "instance subgraphs" that represent a possible execution of a workflow process. Their language, however, does not support cycles and the approach cannot be easily generalized if cycles were allowed. Finally, according to [Low01] Microsoft's BizTalk workflow component (called BizTalk Orchestration) uses a modelling language called XLANG that is based on Milner's π-calculus ([Mil99]), however we were unable to find an article describing the semantics of XLANG in detail using the π-calculus formalism.

Not only is it the case that few proposals for new modelling techniques are accompanied by a formal semantics, it is also rare to find a comprehensive investigation into the requirements for control flow specification in a workflow modelling language and hence our work on workflow patterns appears to be unique in this field. Other 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. Some of the patterns presented in this thesis are related to the control-flow patterns described in [JB96]. However, the goal of [JB96] is to develop a workflow management system that can be extended with new patterns rather than structuring and evaluating existing patterns. Other authors have coined the term workflow patterns but addressed different issues. In [WAH00] a set of workflow patterns inspired by Languages/Action theory and specifically aiming at virtual communities are introduced. Patterns at the level of workflow architectures rather than control flow are given in [MB97]. In [Lon98] Lonchamp presents 13 patterns describing scenarios relatated to collaborative work on one or more documents and the focus is more on the data flow.

1.4 Thesis Structure

This thesis provides both a practical and a theoretical approach to control flow analysis in various workflow modelling languages.

In Chapter 2 we introduce the reader to workflow modelling and we informally present a number of commercially available workflow modelling languages. The aim of this chapter is to demonstrate the great diversity of different modelling approaches being used in industry as well as to provide a future reference for subsequent Chapters. In this Chapter we also present a "hands-on" approach towards a first approximation of a given workflow language's modelling power through a workflow "test harness".

In Chapter 3 we present a practical approach towards identifying requirements for workflow language expressiveness through workflow *patterns*. Patterns address business requirements in an imperative workflow style expression, but are removed from specific workflow languages. We then present a comparison of workflow management systems introduced in Chapter 2 using these patterns. This work is based on [ABHK00] and [AHKB00].

Chapter 4 defines a theoretical framework necessary to establish the *theoretical* limit of a given modelling technique. We introduce the reader to formal specifications of different workflow modelling paradigms as well as provide a formal equivalence notion necessary for comparison of different modelling languages.

In Chapter 5, based on the framework defined in Chapter 4 we establish some fundamental properties of different workflow modelling approaches.

In Chapter 6 we introduce several more advanced modelling concepts and, again based on the foundation established in Chapter 5, we provide important insights into their expressive power and suitability. The work in Chapters 4-6 is based on [KHA01], [KHB00] and [HK99].

Finally in Chapter 7 we present the conclusion of this thesis. The major contributions of the research outcomes are summarized and areas for future research are identified.

Chapter 2

Industry: State of the Art

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 models) are defined to specify which *activities* need to be executed and in what order. 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.

Routing specification requires an adoption of the workflow modelling language. In this Chapter we present modelling languages of several commercially available products. Our aim is to show diversity of different approaches to workflow modelling as well as to provide a reference for the future comparison of different modelling techniques. Before we introduce specific products, we present a common workflow modelling terminology that will be used in this thesis based on the glossary promulgated by Workflow Management Coalition.

2.1 Workflow Management Coalition

The Workflow Management Coalition (WfMC), founded in August 1993, is a non-profit, international organization of workflow vendors, users, analysts and university/research groups.

The Coalition's mission is to promote and develop the use of workflow through the establishment of standards for software terminology, interoperability and connectivity between workflow products. Consisting of over 285 members (as of beginning of 2002), spread throughout the world, the Coalition has quickly become established

as the primary standards body for this rapidly expanding software market. Since its inception WfMC has focused on furthering the field of workflow management by providing standards, common terminology, and interfaces.

Given all that it may be surprising to find, that very few workflow vendors conform to the WfMC recommendations. In fact, very few of them even use a standard terminology, as defined in WfMC basic document "Terminology & Glossary" ([Wor99b]). Despite all these problems it is important to have a closer look at these definitions, not least because they provide a good starting point for more advanced discussion.

According to [Wor99b] a Process Model is a "formalized view of a Business Process, represented as a co-ordinated (parallel and/or sequential) set of process activities that are connected to achieve a common goal". Activity itself is "a piece of work that forms one logical step within a process". According to the definition, it is typically the smallest unit of work which is scheduled by a workflow engine during process enactment. There is very little said about the semantics of an activity, in particular there is no discussion on state transition diagram that is used to manage activity lifecycle.

The activities and other process control constructs are connected using transitions which may be unconditional, such that completion of one activity always leads to the start of another, or conditional, where the sequence of operation depends upon one or more Transition Conditions.

In this thesis we will graphically represent an activity as a rectangle with activity's name in the middle and a transition as an arrow between activities and/or other process control constructs. Apart from activities [Wor99b] recommends that the following process control constructs should be supported by a workflow management system.

AND-Split is "a point within the workflow where a single thread of control splits into two or more threads which are executed in parallel within the workflow, allowing multiple activities to be executed simultaneously". The WfMC additionally observes that "in certain workflow systems all the threads created at an AND-Split must converge at a common AND-Join point (Block Structure); in other systems convergence of a subset of the threads can occur at different AND-Join points, potentially including other incoming threads created from other AND-Split points (Free Graph Structure)". The AND-Split construct is graphically depicted on Figure 2.1.

OR-Split is "a point within the workflow where a single thread of control makes a decision upon which branch to take when encountered with multiple alternative workflow branches". The WfMC does not clearly distinguish between the split in which one and only one path is always taken, and the split in which any number

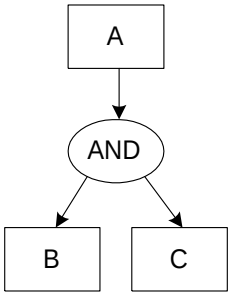

Figure 2.1: Graphical representation of an AND-Split

of paths (possibly zero) is taken depending on conditions associated with each path. As this distinction is important in our thesis, and to avoid any possible ambiguities, we will call an *XOR-Split* a split in which one and only one path is taken (depending on conditions associated with this split). The OR-Split and XOR-Split constructs are graphically depicted in Figure 2.2.

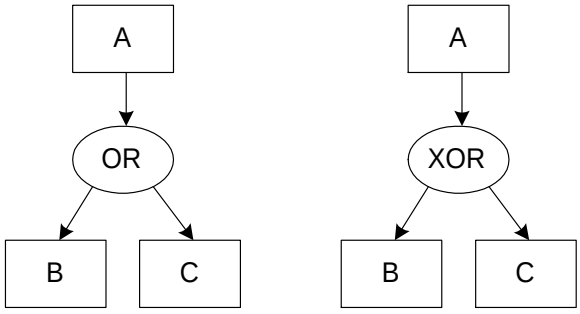

Figure 2.2: Graphical representation of an OR-Split and XOR-Split

AND-Join is "a point in the workflow where two or more parallel executing activities converge into a single common thread of control". Each parallel executing thread is held until the set of all thread transitions to the next activity is completed, at which point the threads converge and the next activity is initiated. In this definition there seems to be an implicit assumption that both parallel threads eventually will "reach" that common point as it is not stated what should happen if this is not the case. The AND-Join construct is graphically depicted in Figure 2.3.

OR-Join is a "point within the workflow where two or more alternative activity(s) workflow branches re-converge to a single common activity as the next step within the workflow". In addition it is noted that "as no parallel activity execution has occurred at the join point, no synchronization is required". Again,

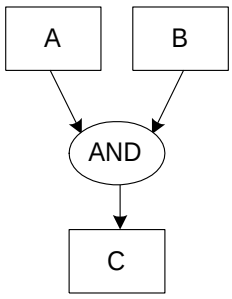

Figure 2.3: Graphical representation of an AND-Join

it is unclear how the OR-Join should behave if parallel execution actually *does* occur before the join point. The OR-Join construct is graphically depicted on Figure 2.4.

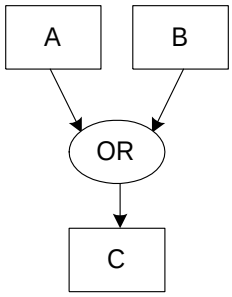

Figure 2.4: Graphical representation of an OR-Join

There are several reasons, we believe, why industry has been unsuccessful in embarking fully on these definitions. The two most important amongst them are:

- The constructs are defined in a very vague and ambiguous manner. Especially in processes which WfMC calls "Free Graph Structures" the interaction between the constructs is far from clear. As an example consider a process in which an AND-Split is followed by an OR-Join. How should this process behave? The WfMC definitions do not provide any hint. In fact, as it is later demonstrated in this thesis, almost every product that we have evaluated took a different view on the semantics of these constructs in more complex scenarios.

- The WfMC, to support the biggest possible range of products takes a "least common denominator" approach. Indeed these basic constructs, or their equivalents, can be found in every single system we have evaluated. However, with

these constructs one can build only the simplest process models. Complex processes require more sophisticated modelling primitives, and this is the area, where, again, products differ in a substantial way.

These issues definitely hamper the WfMC's attempts to provide standards for workflow definition interoperability. If vendors take fundamentally different views on how given products should behave, clearly, process interoperability between them is difficult. What is needed, is a much more thorough insight in the semantics of each product.

2.2 Commercial Products

In this section we will introduce eight different workflow modelling languages supported by commercially available workflow management products. These products have been selected based primarily on their availability and/or willingness of the vendors to cooperate in our research. We have tried to incorporate major workflow offerings such as Staffware, FileNet Visual WorkFlo and MQ Series Workflow. The latter one is of particular interest as it sparked interest in academia (see e.g. [AAA+95, MAGK95]). SAP R/3 Workflow is included as SAP has the largest customer base in the ERP systems market (although we have no data how many SAP implementations actually use its Business Workflow component). Fujitsu's i-Flow is one of the first workflow engines designed primarily for use on the Internet while Verve is one of the first workflow engines specifically designed as an embeddable component in other systems. Finally we have also included Forté Conductor and HP Changengine as they both have interesting modelling concepts and these systems were available to us during the course of this research. When describing each modelling language the focus is exclusively on control flow. The insight in the semantics of each particular language is gained primarily from studying workflow process design manuals that accompany each workflow product. It may be worth noting that none of the evaluated products is accompanied by a specification of the formal semantics of its workflow specification language. That may be understandable as the targeted audience of the manuals are typically process designers without a proper formal background. However, as is the case with the WfMC glossary document, we will see that the informal language used in these manuals makes the workflow semantics ambiguous at times. In subsequent Chapters we will try to present some techniques aimed at a more thorough understanding of a given modelling language's semantics. In some cases it may seem that the versions of products are outdated. On a practical level it is one of the consequences of the relatively long period during which this research was conducted, and we have made the best effort to obtain the latest releases when possible. Nevertheless, it is

our experience that the control flow specification of workflows rarely, if ever, changes with the release of a new version of the product.

To graphically illustrate each product's modelling language we have chosen to show an implementation of a sample process shown in Figure 2.5. This process employs all WfMC-defined basic constructs which are used in a typical context leaving no scope for ambiguous interpretation.

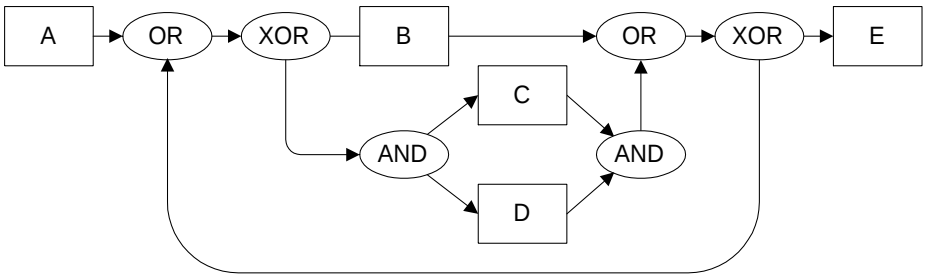

Figure 2.5: Sample process model

2.2.1 FileNet's Visual WorkFlo

Visual WorkFlo [Fil97, Fil99] is one of the leading workflow products in the 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. Our evaluation is based on version 3.0 (introduced in late 1998) of the product.

Visual WorkFlo [Fil97, Fil99] has a relatively simple workflow modelling language. A process in Visual WorkFlo is called an *Instruction Sheet* and it consists of a number of *Steps*. Steps cannot be connected in an arbitrary manner - the graphical process design environment imposes several syntactical restrictions on a process modeller. These are later clarified when a specific Step type is discussed.

The following Step types are available.

Activity is an atomic work unit. It defines what work has to be done in this step of the process and as such, it does not have any additional semantics related to control flow.

Branch is used to model a choice in a process. It directly models a XOR-Split/OR-Join construct combination. A branch can have many outgoing transitions -

however one and only one can be fired depending on the condition defined in a Branch. A Branch always converges in a merge point.

Static Split and Rendez-Vous are used to model a concurrent split in a process. Each Static Split needs to be followed by a Rendez-Vous which converges parallel threads spawned off by a split into a single thread of execution. The combination explicitly models an AND-Split/AND-Join pair.

Goto and Label are used if one wants to bypass certain processing instructions. Using a Goto construct requires the use of a Label construct, and vice-versa. Multiple Goto constructs can reference the same Label instruction. The combination is constrained with many rules and as such it cannot be used to model arbitrary workflows. Specifically, one cannot use a Goto instruction to jump to a different process definition. One can jump from within a Branch to the main process outside the branch, however the opposite is not allowed. One can jump within one branch but not from branch to branch. One cannot jump outside of a static split. One cannot jump into a static split. One cannot jump from one branch of a static split to another, but one may jump within a single branch. Effectively, with so many constraints, each Goto can be easily modelled by a Branch and is primarily used to unclutter more complex branches.

While construct is used to model a structured loop, that is a loop that has only one entry and one exit point.

Release is used to spawn off a branch asynchronously. It can only be put in one (or more) of the branches of the Static Split - Rendez-Vous pair. A Rendez-Vous will only synchronize branches without a Release construct. The spawned-off branches run independently and a subsequent synchronization is not possible.

Terminate is used to prematurely end processing of the workflow. A special sub-process called *Terminate Instruction Sheet* is then invoked to enable some clean-up work.

Call is used to synchronously call another Instruction Sheet (sub-process).

A process has one starting node and one ending node.

Figure 2.6 shows the sample process introduced in Section 2.2 implemented with Visual WorkFlo.

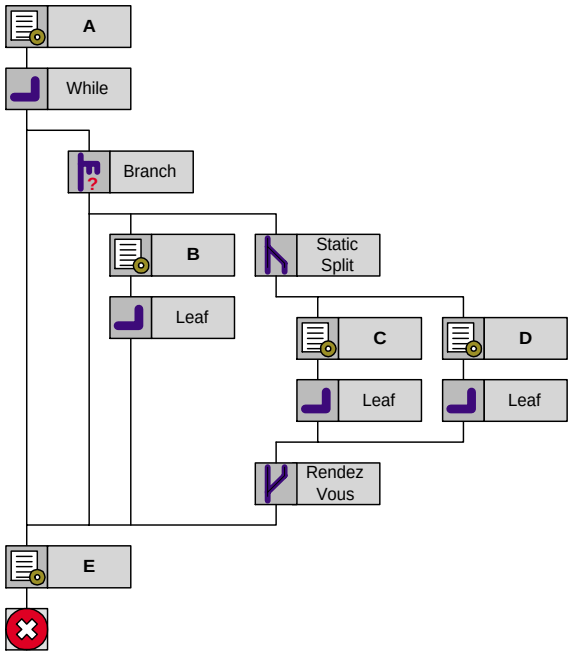

Figure 2.6: Sample process model implemented with **Visual WorkFlo**

2.2.2 Forté Conductor

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 modelling language is powerful and flexible. In October 1999 Forté Software was 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.

In Forté a process definition is laid out graphically as a series of connected activity definitions. Activities are linked to other activities by transitions which are called *process routers* in Forté.

The following activity types are available for the workflow modeller:

Activity represents an atomic piece of work. There are primarily three kinds of regular activities, i.e. offered activities, queued activities and automatic activities. However, from a control point of view there is no semantical difference between the three as they only differ in the way they are offered to the users.

Each activity has an associated *trigger condition*. There are three types of trigger conditions:

1. Trigger when any router arrives
2. Trigger when all routers arrive
3. Custom trigger

The trigger condition defines the semantics for an activity's behaviour primarily in a situation when the activity has more than one incoming transition. If an activity has a trigger of type "all routers arrive" it will act as an AND-Join. If an activity has a trigger of type "any router arrives" it will act as an OR-Join. With help of a custom trigger one can write a condition using a process definition's variables or a special *count* variable which indicates how many times any other arbitrary activity in the process has executed.

An activity can have more than one outgoing transition (router). In that case, Forté evaluates a condition associated with each transition. Depending on the activity's setting it can either:

1. Evaluate conditions of all routers and trigger these routers for which the conditions evaluated to "true" (possibly none).
2. Evaluate conditions of each routher in a sequence until the condition of some router evaluates to "true". This will be the only router to be triggered. Again, it is possible that no router is selected.

One can also specify a special *else* router which will execute if all other router conditions evaluate to false. If all *OnComplete* router methods return False, the engine takes no further action for that activity. Thus an activity with more than one outgoing transition can model either an AND-Split or an OR-Split.

There are no syntactical restrictions governing the use of activities and process routers. Thus, any arbitrary cycles are possible to model.

It is possible to have many concurrent instances of one activity running at the same time.

First Activity is a special activity type that makes a starting point of a process instance. There can be only one First activity in a process. There is no extra semantics associated with this activity type.

Last Activity is a special activity type that causes the termination of a process instance. There can be only one Last activity in a process.

Junction Activity has no real associated semantics. It is primarily used to unclutter more complex diagrams.

Subprocess Activity represents a subprocess definition. A subprocess can be instantiated either synchronously or asynchronously depending on its settings.

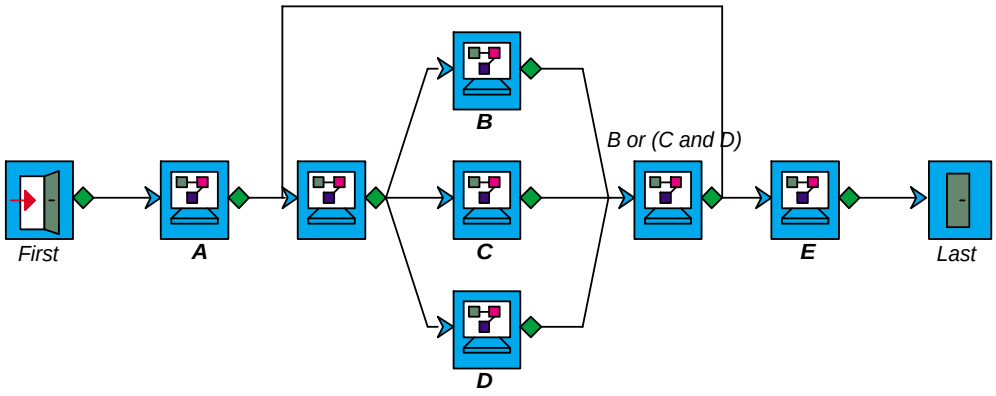

Figure 2.7: Sample process model implemented with **Forté Conductor**

The sample process introduced in Section 2.2, implemented with Forté Conductor, is shown in Figure 2.7. Note the need for using *null* activities, i.e. activities that have no task associated with them. One of these activities, preceding activity E has a *custom condition* defined so that it triggers when either B or both C and D complete. Furthermore, conditions associated with the output routers of the null activity subsequent to activity A are such that either B or C and D are fired. Both triggering condition and conditions associated with output routers are implicit in a diagram making a quick interpretation of the diagram semantics by end-users impossible.

2.2.3 Changengine

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, has been introduced in 1998 and it is 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.

A business process in Changengine consists of a number of different process elements called *Nodes* linked by *Arcs*.

The following is a summary of nodes available for process modelling.

Work Node is a node that defines some work to be done. Once a work node has been executed, it cannot be executed again unless it is reset. This interesting behaviour is explained in more details later on when the *Reset Arc* is introduced. This semantics prevents the process from having multiple instances of the same

node running concurrently. Syntactically, a work node can have only one input and one output.

Complete Node is situated at the end of each branch of a process where no further nodes are to be executed. This node has no associated semantics, it simply visualises the end of a process. A complete node can have only one input, and has no outputs.

Start Node is the starting node of a process. It can be seen as a special type of a Work Node, as it also defines some work to be done.

Route Node is used to split a link to two or more destinations, merge links together, or both. A Route Node with more than one input is called a *Merge Node*, a Route Node with more than one output is called a *Split Node* (a route node can be both a split and a merge).

A Merge Node can be synchronising or non-synchronising depending on the rule associated with that node. For example, a route rule *"If WorkNode1 = COMPLETED AND WorkNode2 = COMPLETED THEN Goto WorkNode3"* states that $WorkNode3$ can be started only when both $WorkNode1$ and $WorkNode2$ are complete. Similarly, one may define a rule such that only one or some of the incoming branches need to be complete before the next activity following the merge is performed.

A Split Node can fire any combination of outgoing arcs depending on rule conditions defined for this node.

A Route Node must have at least one input and at least one output.

Abort Node is used to terminate a process. Process terminates immediately, even if other nodes in the process have not yet completed. An abort node has exactly one input and no outputs.

There is no special graphical construct used for decomposition, however, when defining a Work Node, a process designer might specify another process instead of a simple task to be performed. This way one can achieve decomposition in an implicit manner.

A process terminates once all the end points have been reached.

Loops and Reset Arcs

In Changengine one may draw a loop in an arbitrary way. Because a Work Node that has been executed cannot be executed again, to be able to use a Work Node in a loop, a special construct, called a *Reset Arc* has been provided. A Reset Arc is a link that

loops back to an earlier node in a process, possibly causing one or more nodes to be executed again. All work nodes on a directed path between the end and the beginning of a Reset Arc are said to be in-scope of this Reset Arc. When the Reset Arc is fired, any in-scope work nodes are initialised so that they can be run again. Moreover, if there are still any running nodes within the scope of the loop (as could be the case if an AND-Join is followed by a non-synchronising merge) then the running nodes will be stopped and reset.

Some interesting scenarios are possible. Consider a scenario in Figure 2.8 which depicts a loop with multiple entry points.

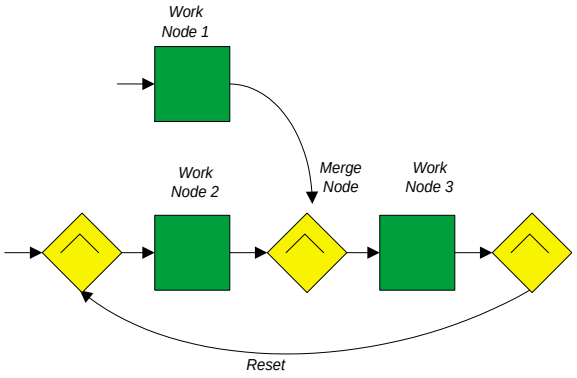

Figure 2.8: Loop with multiple entry points in Changengine

If the loop is activated and the merge node is non-synchronising, the process can advance through the merge node from Work Node 2 without having to wait for Work Node 1 to complete.

However, if the merge node is synchronising and the loop is activated, the merge node will wait for Work Node 1 to complete again. Unless Work Node 1 has also been reset, the process will deadlock.

Another common scenario is a loop with multiple exit points as shown in Figure 2.9.

When the reset arc is fired, Work Node 3 is not reset because it is not within the scope of the loop, and therefore cannot be run again. When the split node is reached again because one of the branches loops back, other branches outside the loop cannot be performed a second time.

The sample process introduced in Section 2.2 implemented with Changengine is shown in Figure 2.10

2.2.4 Staffware

Staffware [Sta00] is one of the leading workflow management systems. Staffware

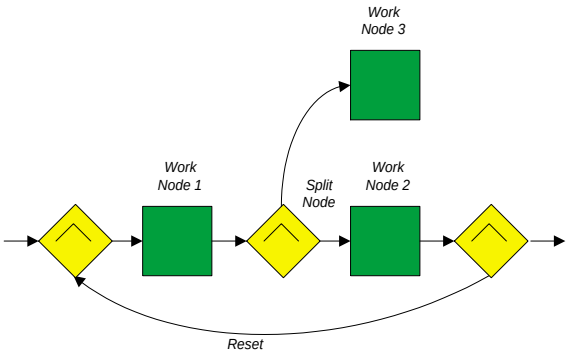

Figure 2.9: Loop with multiple exit points in Changengine

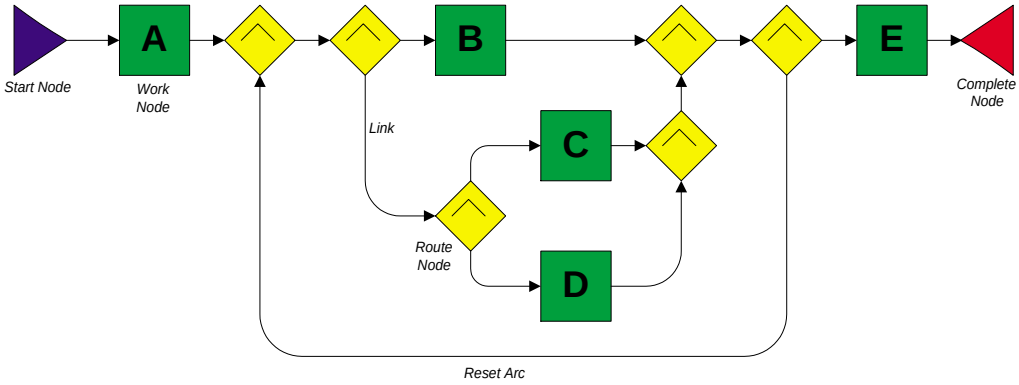

Figure 2.10: Sample process model implemented with **Changengine**

is authored and distributed by Staffware PLC. Staffware PLC has its headquarters in Maidenhead (UK), operates through offices in 15 countries and has a network of 360 partners, resellers and OEMs. We used the most recent version of Staffware (i.e. Staffware 2000), which was released in the last quarter of 1999. In 1998, it was estimated by the Gartner Group that Staffware has 25 percent of the global market [Cas98].

A workflow definition in Staffware is called a *Procedure*. A procedure consists of a number of *Objects* connected with *Lines*. Linking Objects with Lines defines control flow - a source Object of the Line has to complete before the destination Object of the Line can be started. The following Objects are supported:

Step corresponds to an activity. It is depicted by a form icon. Staffware differentiates between normal steps, automatic steps, event steps and open client steps. This differentiation has no effect on the control semantics of the process. A Step can be linked to many other Steps. The underlying semantics is that of an AND-

Split. Several Steps can link to one Step. The underlying semantics is similar to an OR-Join, but Staffware does not support concurrent multiple instances of a Step in a running state. A new instance of a Step will override any existing instances.

Start indicates the beginning of the process and is depicted as a green street light icon in the Procedure diagram. There can be only one Start object in a Procedure. A Start can have only one single Line linking it with another Object in the Procedure. There is no semantics defined for Start - it is only used as a visual representation of the beginning of the flow.

Stop indicates the end of a process. It is represented by red street sign icon. It does not have any underlying semantics and it is only used as a visual representation of the end of the flow. A Procedure can have many Stop objects. At most one Object can link to a Stop.

Condition object models a binary XOR-Split. It is represented in a graph by a diamond icon with a question mark. There is one logical condition associated with the Condition object. The Condition has only one incoming transition and two sets of outgoing transitions. - one set is triggered if condition evaluates to "true", the second if condition evaluates to "false".

Wait is used for synchronisation of parallel paths and is represented by an hourglass icon in the diagram. A Wait object has a semantics similar to that of an AND-Join. It has two types of incoming transitions. A first type (represented by a solid line) links a Step that will "wait" for other Steps to finish before the process can continue. There can be only one such transition. Transitions of the other type (represented by dashed lines) link Wait to "waited for" Steps. There can be many such transitions (see Figure 2.11).

Router is a visual object that improves the design of the process. It is represented by a small circle and it has exactly one incoming and one outgoing transition. Router does not have any associated semantics.

Objects in a process diagram are connected by solid black lines that correspond to transitions. There can be only one such Line between two Steps. It is possible to draw a special Line between Steps called *Withdrawal Connection*. If a Step that is a source of a Withdrawal Connection is started, Staffware will attempt to remove any existing instance of a Step that the withdrawal connection points to. Withdrawal Connections can only point to normal Steps.

Staffware does not implement decomposition directly. Decomposition can be achieved by instantiating a new process as part of the activity processing. Synchronisation between the main process and a subprocess can be achieved using events.

There is no restriction in Staffware as far as loops or cycles in the workflow graph are concerned.

Figure 2.11 shows the sample process introduced in section 2.2 implemented with Staffware.

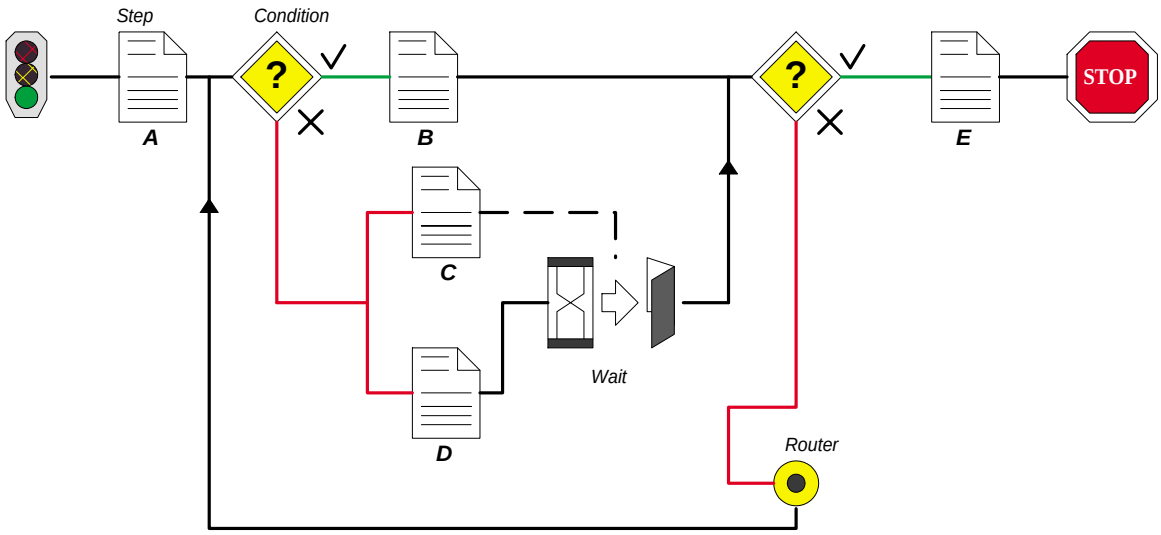

Figure 2.11: Sample process model implemented with **Staffware**

2.2.5 Fujitsu i-Flow

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

An i-Flow process definition consists of a number of nodes connected by arrows (representing transitions) in an arbitrary way with very few syntactical restrictions. The following are the node types available for workflow modelling:

Activity Node represents an atomic step in a workflow process. Upon receiving an event from any of the incoming arrows, an activity node enters a running state. While in this state, it ignores any events that it receives from its incoming arrows. If an activity has more than one outgoing arrows, an assignee of a work

item associated with this activity node chooses one of the outgoing arrow for the process to continue.

Start Node is used to indicate the start of a process. All nodes following a start node are immediately instantiated. A process can have only one Start Node. A Start Node cannot have any incoming arrows.

Exit Node marks the end of a workflow process. If a process has multiple exit nodes, once the first one is reached, process is terminated. An Exit Node cannot have any outgoing arrows.

Conditional Node represents a step in a workflow process where the process flow proceeds to one of the outgoing arrows depending on process data. Upon receiving an event from one of the incoming arrows, the conditions are evaluated, for the first one with condition evaluated to "true" an event is sent. If all conditions evaluate to "false", an event is sent to an arrow specified as *default*. The Condition construct is a direct equivalent of the XOR-Split.

AND Node represents a step in a workflow process where the process pauses to synchronize multiple threads of execution (AND-Join). An AND Node waits till it receives at least one event on each of its incoming arrows. Any additional event received from the same arrow is ignored while waiting to receive events from each of the other arrows. When each of the incoming arrows have received an event, all execution threads are synchronized and events are sent to all outgoing arrows.

OR Node represents a step in a workflow process from where a single thread splits into multiple parallel threads. The node can have many incoming arrows. Every time it receives an event from one of its incoming arrows, it sends an event to all its outgoing arrows. OR Node can be used as both OR-Join as well as AND-Split.

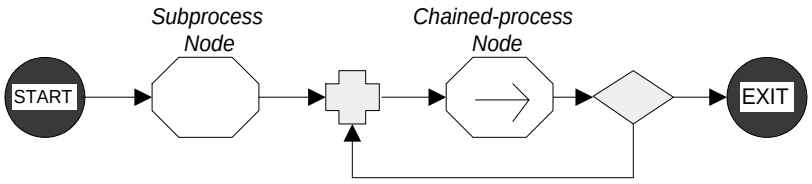

Figure 2.12: Subprocess definition in i-Flow

Decomposition is modelled using two special node types, subprocess nodes and chained process nodes.

Subprocess Node represents a separate process definition. On receiving an event from one of the incoming arrows a subprocess node starts a separate instance of its subprocess and waits for it to complete. While waiting, it ignores any events received from the incoming arrows. Upon completion, the subprocess sends an event to all its outgoing arrows.

Chained-process Node also represents a separate process definition. Unlike the subprocess node it does not wait for the new process to complete and immediately after instantiating a new subprocess instance it sends an event to all its outgoing arrows.

Both nodes are depicted in Figure 2.12.

The sample process introduced in Section 2.2, implemented with i-Flow is shown in Figure 2.13.

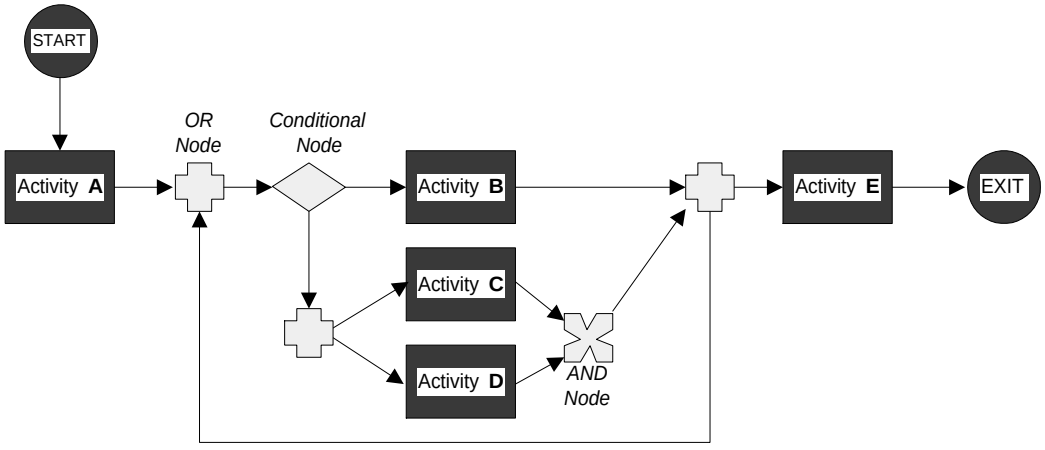

Figure 2.13: Sample process model implemented with **i-Flow**

2.2.6 MQSeries Workflow

MQSeries Workflow [IBM99] 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 move from the proprietary middleware to middleware based on the MQSeries product. Our evaluation is based on version 3.1.

The terminology of MQSeries Workflow is very consistent with the recommendations of the Workflow Management Coalition. A business process model consists of a num-

ber of activities linked together by *connectors* representing transitions. The activities can be connected in an arbitrary manner as long as there are no cycles.

There are three different types of activities:

Program Activity represents a task to be done in order to complete a business process. Each program activity has an executable application associated with it that will be used by the end-user to perform that task.

Block Activity is a construct to model structural decomposition of a process. Similarly to a top-level process it consists of a number of activities. Recursive block execution is impossible since every object within the block is visible only in this block's namespace.

Process Activity is similar to Block Activity. The difference is that the definition of the Process Activity is contained within a separate process object. That means that subprocesses can be shared between many different processes (otherwise the semantics of the Process Activity is identical to that of a Block Activity).

Every connector in an MQSeries Workflow process model has a condition associated with it that is evaluated during process execution. Each activity has a field indicating whether it will be started if at least one incoming connector evaluates to "true", or it will be started if all incoming connectors evaluate to "true" (the setting of this field in an activity with only one incoming connector is irrelevant).

An activity can only be started if all of its incoming connectors are evaluated and the condition associated with it is met. If all connectors are evaluated and the condition associated with it is not met (for example all connectors evaluate to False for an activity requiring at least one of its connectors to evaluate to True), it is marked as "skipped" and all its outgoing connectors are evaluated to False regardless of their associated conditions. If an activity can be started and it is subsequently completed, all of its outgoing connectors get evaluated.

A business process finishes when all activities in the graph are either completed or marked as "skipped".

Applying the WfMC's definitions to MQSeries Workflow we have that an activity with more than one outgoing connector has an OR-Split semantics, whereas an activity with more than one incoming connector has an OR-Join or an AND-Join semantics depending on the context.

In addition to a condition governing the start of an activity, each activity in MQSeries Workflow also has an *exit condition*. This condition is evaluated once the activity is completed. If it evaluates to True, the outgoing connectors are evaluated and the flow

may proceed to the next activities. If it evaluates to False, the activity is restarted. The exit condition for program activities is typically set to True.

The common use of exit conditions is in association with block activities. As direct cycles are not allowed in MQSeries Workflow, to model loops one has to use a block activity - after the block completes, and the exit condition evaluates to False, the block is restarted.

MQSeries Workflow, in versions prior to 3.1 used to provide one more additional control flow concept called *Bundle*. As this concept is an interesting one (it is surprising that IBM has decided to drop it from subsequent releases) we will introduce it briefly here. Bundle is implemented externally so it does not have a separate graphical representation. The concept behind it is to allow users to implement processes needing multiple instances. A Bundle consists of a *Planning Activity* followed by a Block Activity. The Planning Activity is used to specify during runtime how many instances of the following block are to be instantiated. That value is then used to instantiate the appropriate number of Block Activities in parallel. Such a block completes when every instance of it is completed.

Figure 2.14 shows the sample process introduced in Section 2.2 implemented as an MQSeries Workflow process model. Note that since cycles are not allowed in a process model, a loop has to be modelled as a repetitive block structure. The conditions associated with the transitions from the null activity in the block structure to activities B, C and D have to be such that either only B or both C and D can be performed.

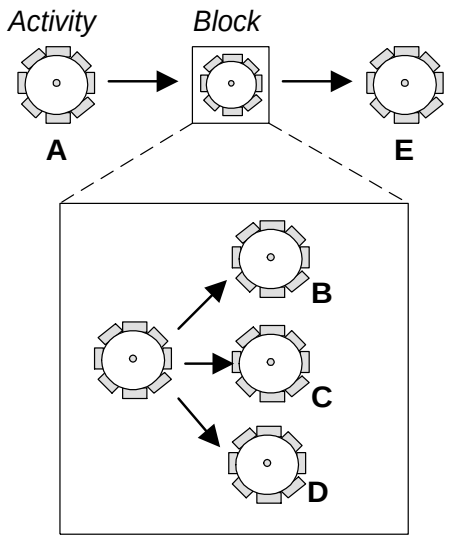

Figure 2.14: Sample process model implemented with **MQSeries Workflow**

2.2.7 Verve

Verve ([Ver00]) is a relative newcomer to the workflow market as it made its debut in 1998. In late 2000 was acquired by Versata and renamed Versata Integration Server (VIS). Our evaluation is based on version 2.0 of the product. What makes it an interesting workflow product is that it has been designed from the ground up as an embeddable workflow engine (i.e. a workflow engine that can be embedded in other applications rather then used as a stand-alone product).

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 and other control flow constructs connected in an arbitrary manner by transitions. Each transition has an associated condition which is independently evaluated. When a condition evaluates to "true", an event trigger is sent to a subsequent construct.

The following basic control flow constructs are available:

Activity represents an atomic unit of work. An activity gets activated as soon as it receives a trigger from *any* of its incoming transitions. Unlike many other workflow products there may be several activity instances running concurrently. An activity with many outgoing transitions represents an OR-Split (or and AND-Split) depending on conditions associated with these transitions. An activity with many incoming transitions implements an OR-Join.

Start Node marks the beginning of a process and does not have any semantics associated with it. There can be only one start node in a process.

End Node marks the end of a process. There can be many end nodes in a process and the first one reached terminates execution of the process.

Synchronizer is used to synchronize many parallel threads of execution into one thread. Once one of the incoming transitions fires, the synchronizer enters a "waiting" state and waits for other transitions to fire whilst ignoring any additional triggers from the transition that has already been "fired". Once all transitions have fired, subsequent activities are triggered. The synchroniser directly implements the AND-Join.

Discriminator has many incoming transitions. Similarly to the synchronizer, once one of the incoming transitions fires, it enters a "waiting" state and waits for the other transitions to fire. In contrast to the synchronizer, it activates subsequent activities as soon as it enters the waiting state. Once all incoming transitions have fired, the discriminator resets itself and can be fired again.

Subprocess represents a subflow that is instantiated synchronously.

The sample process introduced in Section 2.2, implemented with Verve is shown in Figure 2.15. The conditions associated with transitions from the null activity to activities B, C and D have to be such that either B or both C and D are executed.

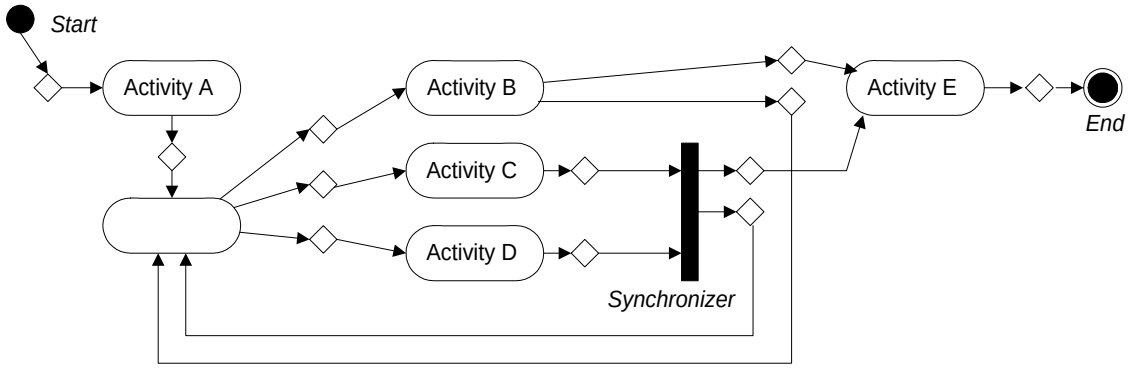

Figure 2.15: Sample process model implemented with **Verve**

2.2.8 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 called SAP R/3 Workflow [SAP97, KT98] that we have evaluated independently of the rest of R/3. Our evaluation is based on release 4.6 of the product which is a current release as of the beginning of 2002.

The SAP R/3 Workflow model is based on an extended version of the EPC modelling technique (Event-driven Process Chains), however the technique is adapted for modelling executable workflows. In comparison to standard EPCs, several syntactical restrictions have been introduced to impose on a workflow processes a structured form. SAP R/3 Workflow Builder allows the designer to switch back and forth between three different graphical views, Classic EPC being one, the SAP R/3 Workflow simplified graphical representation being the second one and a third notation, called simply EPC, which is a cross between Classic EPC and simplified graphical notation of SAP. Throughout this section we will use the simplified graphical representation as it is less cluttered than classic EPC notation.

A workflow model in SAP has exactly one starting and one ending point and comprises the following modelling primitives which are called *steps* in SAP R/3 Workflow :

Task represents either an atomic work unit (single-step task) or it represents a sub-workflow (in which case it is called a multi-step task). A task always has one incoming and one outgoing connector.

An interesting feature of SAP R/3 Workflow is that the actual work to be done as part of a task can be determined during run-time.

When defining a task, apart from the regular processing of a program associated with that task, one can set up a special kind of processing called *Table-Driven Dynamic Parallel Processing*. This allows multiple instances of that task to be running concurrently at the same time. Once all of them are completed, the workflow resumes with the subsequent step.

Condition represents branching in a workflow definition. It has one incoming connector and two outgoing connectors. The workflow system evaluates the condition comparing elements of the workflow container with constants, system fields or other, user-defined fields. The condition evaluates either to True or False and one of the outgoing branches is taken. Both branches of the condition have to merge in a *merge* point. The Condition corresponds to an XOR-Split.

Multiple Condition is very similar to a Condition, however it may have more than two outgoing branches. Only one of the branches can be chosen at runtime, and this is done comparing the value of a given workflow field with a set of predefined constants. One branch can be defined as *Others* and it is taken when the value of the field does not match with any of the constants associated with the branches of the Condition. When there is no Others branch and no match is found, the workflow assumes the *error* status. Multiple Condition corresponds to XOR-Split with more than two outgoing transitions.

User Decision is very similar to *Multiple Condition* but the choice of the outgoing branch is done by the workflow user rather than automatically by the workflow system using some workflow data. From a control point of view there is no semantical difference between *User Decision* and *Multiple Condition*.

Until Loop is a structured loop that is executed until a condition defined for the loop is met. There can be only one entry and one exit point for such a loop.

While Loop is a construct that semantically can be thought of as a combination of a *Multiple Condition* and an *Until Loop*. A While Loop can have several branches, each branch is associated with a certain constant which, at runtime, is compared to a given workflow data field (similarly to *Multiple Condition*). The branch that matches the selection criterion is then chosen and executed. Once it is finished, the condition is checked again. The loop is finished if none of the branches match the selection criterion.

Fork and Join are used to facilitate parallel processing. Each *Fork* has to be associated with a *Join*. The workflow modeller specifies the number of branches being waited for. This is typically equal to the total number of branches in the *Fork* but can be smaller. In the latter case, the workflow can continue when the specified number of branches are finished. The remaining branches of the *Fork* are not allowed to be completed - they are marked as *logically deleted*. It is also possible to specify an additional condition that needs to be met before the workflow can continue beyond the *Join* point. The Fork and the Join can be used to model the AND-Split and the AND-Join respectively.

Process Control is a special activity type that can be configured to execute several predefined commands. Of particular interest to the control flow execution of the workflow process are the following:

- *Cancel Work Item*. It forces another work item of the same workflow into the status *logically deleted*. This essentially completes this other work item and subsequent steps of this work item are not executed. This function can only be used if the process control step and the step to be cancelled are located in a fork.
- *Terminate the Workflow* terminates the current workflow. If there are any incomplete work items, they are forced into the status *logically deleted*. If the terminated workflow is a subworkflow, execution of the superworkflow continues as normal.
- *Cancel Workflow* forces the enactment system to cancel further execution for the current workflow. Incomplete work items are forced into the status *logically deleted*. In contrast to *Terminate the Workflow*, if the cancelled workflow was a subworkflow, the execution of the superworkflow is not continued.

The sample process introduced in Section 2.2, implemented with SAP R/3 Workflow is shown in Figure 2.16.

2.3 Test Harness

Having an existing workflow engine in hand it is impossible to fully derive its semantics in a formal manner. One can only assume that the given language has a particular semantics through the study of the accompanying manuals and the subsequent confirmation of the described behaviour with actual runs of test cases. Such semantics can be called an *observational* semantics and is clearly limited by the fact

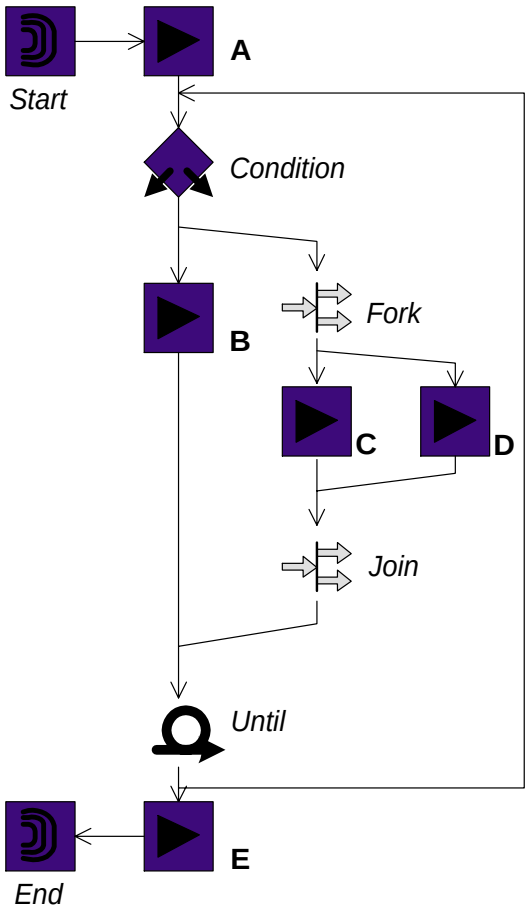

Figure 2.16: Sample process model implemented with **SAP R/3 Workflow**

that it is confirmed through running a finite number of test cases. It may happen (and it has happened to us on few occasions) that the next test case invalidates existing assumptions for the behaviour of some control flow constructs and one needs to come up with an alternative explanation of the observed behaviour. This situation is quite familiar to theoretical physicists who try to establish mathematical models of the universe which are valid until a new experiment is conducted and the results cannot be explained by the existing model. A subsequent theoretical model will then be typically developed in a never-ending pursuit of an ideal model that can explain every physical experiment ever held in the past and to be held in the future.

With software the situation is even more complicated as the evaluated product, as well as the environment (e.g. operating system) in which the workflow software runs, may contain various errors that may cause the software to behave unexpectedly for certain test cases.

Bearing that in mind, we believe that from a theoretical point of view, much more important is the *intentional* semantics of the authors of the product rather that *observational* one. In other words, we are more interested how the product should behave, rather then how it actually behaves. Assuming bug-free implementation, the two semantics should be identical. Unfortunately, in the majority of cases, we do not have access to the authors of the product, so we are left with the often-ambiguous documentation and the product itself to find out how a particular construct behaves.

In this section we would like to propose a set of test cases (process models) that aim at discovering the behaviour of a given workflow product. They can be used to augment the knowledge gained from studying the process design manuals and are designed to expose behaviour that is rarely described in manuals but nevertheless worth understanding. It should be stressed that in theory no amount of test cases will reveal all the semantics of any given workflow language. However, it is easy to add new test cases to the "test harness" and for the products we have evaluated we believe that our proposed test harness is comprehensive enough to reveal important differences between products that will be subsequently analysed in the remainder of this Thesis.

2.3.1 Basic Assumptions

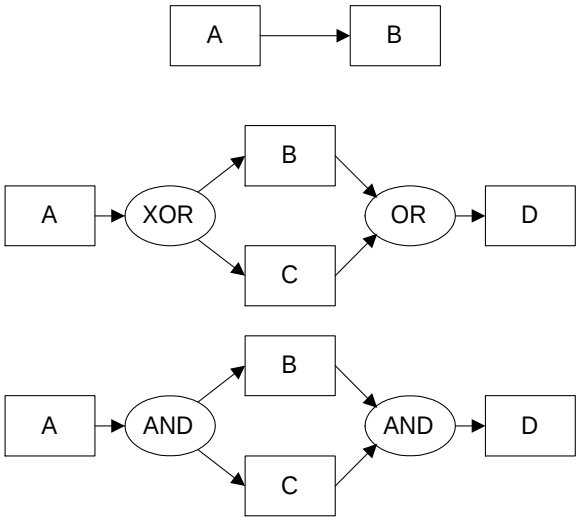

Figure 2.17: Standard processes with well-understood semantics

Every workflow product that we have evaluated implements the WfMC-recommended control-flow constructs and the behaviour of these constructs in the simplest scenarios is identical across the whole product range. These scenarios are shown in Figure 2.17

and the observed behaviour of AND-Split, XOR-Split, AND-Join and OR-Join is consistent and is as follows:

- In the first scenario two activities are joined with a transition. Activity on the "receiving" end of a transition cannot start before the preceding activity completes.
- In the second scenario an XOR-Split is followed by an OR-Join. The observed behaviour is that only one path of the branch is taken, i.e. either activity B or activity C is started. Once the activity on a selected branch completes, activity D can be started.
- In the third scenario an AND-Split is followed by an AND-Join. After completing activity A both activities B and C are started. Once they both complete, activity D can be started.

Each workflow product has a slightly different strategy when implementing the basic control-flow constructs. For example Verve Workflow does not have an explicit OR-Join construct. To model an XOR-Split we have used an activity with two outgoing transitions and we have assigned the conditions to these transitions so that only one of the outgoing transitions can be fired at any one time. Similarly, to implement an AND-Join we have used a Rendez-Vous construct in Visual WorkFlo, a Join in SAP R/3 Workflow, etc... This is the approach that we have used to implement the more complex scenarios presented in the remainder of this Chapter.

2.3.2 Advanced Control Flow

Once it is known in a given workflow language how to implement the basic control-flow constructs, it is interesting to understand what happens when an XOR-Split is followed by an AND-Join and an AND-Split is followed by an OR-Join as shown in Figure 2.18.

Not every workflow product allows the workflow designer to model these two processes. Some products, such as Visual WorkFlo or SAP R/3 Workflow impose stringent syntactical restrictions that force the designer to follow each XOR-Split with an OR-Join and each AND-Split with an AND-Join.

For products that allow these processes to be modelled, in a scenario with an XOR-Split followed by an AND-Join (top diagram of Figure 2.18) we have observed the following different behaviour:

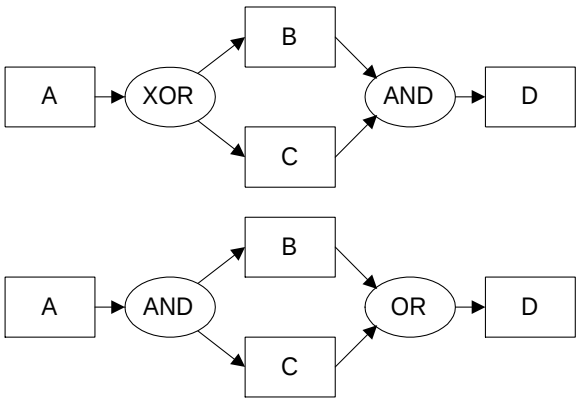

Figure 2.18: Mixed Split/Join constructs

1. Once activity B or C is completed the process deadlocks, activity D will never be started. The only way to resume work is to kill the process instance through an administrator console. This behaviour is more common and can be observed in Staffware, Verve, Forté, i-Flow and Changengine.

2. Once activity B or C is completed, the process is considered to be completed. Activity D never gets executed. This behaviour can be observed with MQSeries Workflow.

The scenario in which an AND-Split is followed by an OR-Join (bottom diagram of Figure 2.18) is more complicated as the observable behaviour in some products depends on the execution sequence of the activities. Let us assume that we always complete activity B before activity C (it should not matter as the process is symmetrical). The following different behaviour has been observed:

1. Once activity B is completed, an instance of activity D is intantiated and can be started. When activity C is completed before the instance of D is started, a second instance of activity D is created and put on the user's worklist. Two instances of D can then be started in parallel. This behaviour can be observed with Forté and Verve.

2. Once activity B is completed, an instance of activity D is created, and can be started. If this instance is completed before activity C completes, a second instance of activity D is created. However if the first instance of activity D is not completed before activity C completes, a second instance of D will not be created. This behaviour can be observed with Staffware and i-Flow.

3. Once activity B is completed, activity D can be started. There will be only one instance of activity D regardless of the state of activity C. This behaviour can be observed in Changengine.

4. Activity D cannot be started before both activities B and C are completed. This behaviour can be observed in MQSeries Workflow.

Table 2.1 provides a summary of the behaviour for the scenario involving an XOR-Split followed by an AND-Join.

Product	Behaviour
MQSeries	process terminates after executing either B or C
Forté	process deadlocks after executing either B or C
Verve	process deadlocks after executing either B or C
Changengine	process deadlocks after executing either B or C
i-Flow	process deadlocks after executing either B or C
Staffware	process deadlocks after executing either B or C
Visual WorkFlo	not allowed
SAP	not allowed

Table 2.1: Test result for a process containing an OR-Split followed by an AND-Join

Table 2.2 provides a summary of the behaviour for the scenario involving an AND-Split followed by an OR-Join.

Product	Behaviour
MQSeries	D executed once, after both B and C completes
Forté	D executed twice, once after either B or C finishes
Verve	D executed twice, once after either B or C finishes
Changengine	D executed once, after either B or C finishes
i-Flow	D executed once or twice, depending on execution scenario
Staffware	D executed once or twice, depending on execution scenario
Visual WorkFlo	not allowed
SAP	not allowed

Table 2.2: Test result for a process containing an AND-Split followed by an OR-Join

2.3.3 Termination

The next two scenarios are used to test the termination policy of the workflow engine. If the workflow language allows the process to have more than one final task (i.e.

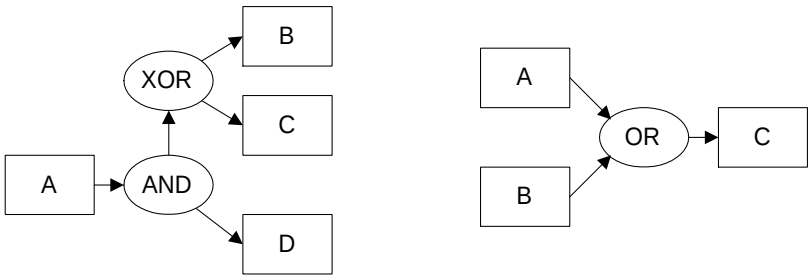

Figure 2.19: Termination policy scenarios

a task without any outgoing transitions), it is important to understand when the process will be considered terminated by the workflow engine.

The scenarios are depicted in Figure 2.19. The left diagram can be used with any workflow engine that allows more than one final task. The right diagram is to be used with workflow engines that will instantiate activity C twice and the goal is to find out if the workflow engine will wait for both instances of C to terminate before it considers the workflow to be complete.

With the left hand diagram the following behaviour has been observed:

1. The process is terminated once either activity B, C or D is completed regardless of the state of the other activities. This can be seen with Verve, Forté and i-Flow.

2. The process is terminated after all activities are completed, i.e. either B or both C and D. This behaviour can be seen with MQSeries Workflow and Staffware.

3. The process never completes (as it waits for all final activities to complete). As that can never happen (either B or C can complete but not both) the process deadlocks. This can be seen with Changengine.

Table 2.3 provides the summary of the behaviour for both termination scenarios:

2.3.4 Multiple Instances

The test shown in Figure 2.20 is designed to test the behaviour of an AND-Join when used in a multiple instances scenario. Ideally one would expect activity D to be executed twice regardless of the execution scenario of the remaining activities. In some languages though that is not the case as the synchronizer is not commutative (it does not remember the triggers that fired it). The behaviour we observed in Verve is the following: Assume an execution scenario in which activities A-D are completed

Product	Left diagram	Right diagram
MQSeries	Both B and C complete	not allowed
Forté	Either B or C complete	only one C can complete
Verve	Either B or C complete	only one C can complete
Changengine	Process deadlocks	not allowed
i-Flow	Either B or C complete	only one C can complete
Staffware	Both B and C complete	both C can complete
Visual WorkFlo	not allowed	not allowed
SAP	not allowed	not allowed

Table 2.3: Test result for a process containing multiple termination points

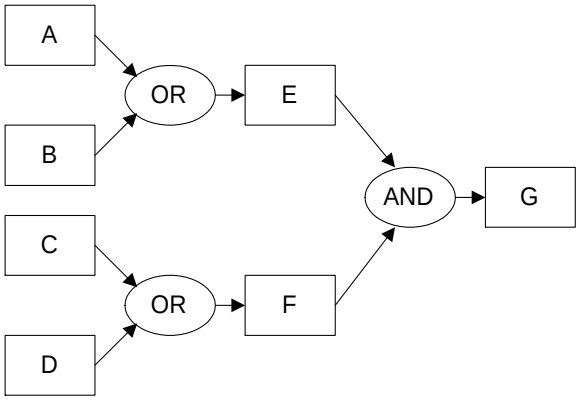

Figure 2.20: AND-Join with multiple instances

and there are two instances of E and two instances of F ready for execution. Once one of the instances of E is completed, the AND-Join (Synchronizer in Verve) gets instantiated and waits for an instance of F to complete. While in the "waiting" state, it ignores any other triggers from E, so when the second instance of E completes before the first instance of F does, the extra trigger is lost. When an instance of F eventually completes, G is started. When the second instance of F completes, the AND-Join is instantiated and waits for an instance of E. That is never going to happen though as both instances of E are already completed, thus the process will deadlock. The careful reader will notice that a slightly different execution scenario (an instance of E followed by an instance of F followed by another instance of E and F will result in instantiation of two instances of activity G).

Staffware for this scenario behaves slightly differently as the AND-Join construct in Staffware is not symmetric. 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, whereas multiple signals from the latter type of transition are remembered.

2.4 Summary

In this Chapter we have presented basic control flow modelling constructs based on definitions by Workflow Management Coalition. Then we have introduced eight different workflow modelling languages from the leading workflow vendors. Vagueness of WfMC definitions provided workflow vendors with opportunity to implement these workflow constructs using different approaches. The differences between semantics of common workflow constructs are highlighted using a set of simple business process which, when implemented in different workflow engines, behave differently. These examples provide strong empirical evidence that to achieve workflow interoperability and business process interchange one need to consider standardization of both process syntax and process semantics. Moreover, the semantics of process constructs should be formal so that there is no room for different interpretations on the workflow vendors part.

Chapter 3

Workflow Patterns

Even without formal qualification, the distinctive features of different workflow languages allude to fundamentally different semantics. As we have shown in Chapter 2 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 Chapter, 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. Some of the basic requirements identify slight, but subtle differences across workflow languages, while many of the more complex requirements identified in this Chapter, in our experiences, recur quite frequently in the analysis phases of workflow projects, and present grave uncertainties when looking at current products.

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. As a whole, we show that the workflow patterns are comprehensive with respect to current workflow languages. Indeed, no contemporary workflow management system supports all the patterns. For those patterns that were supported, some had a straightforward mapping while others were demonstrable in a minority of tools.

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

It is important to understand that all 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. This is a pragmatic approach which is taken to avoid unnecessary, lengthy discussions related to semantical subtleties. In Chapter 4 we will take a more rigorous approach when we will present a formal foundation for establishing workflow modelling language expressiveness.

The systematisation of the workflow patterns is a result of the collaborative work between the author of this thesis and his supervisor, Arthur ter Hofstede, Alistair Barros of DSTC, and Wil van der Aalst of University of Eindhoven. I would like to especially acknowledge Wil van der Aalst for his initial input as well as providing an insight to state-based patterns presented in section 3.5.

3.1 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 [Wor99b]. These patterns are re-stated here for completeness and reference.

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 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.

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

- Activity *Evaluate Claim* is followed by either *Pay Damage* or *Contact Customer*.

- Based on the workload, a processed tax declaration is either checked using a simple administrative procedure or 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.

□

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 OR-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 behaviour). In other languages workflow designers themselves are responsible for the design not having the possibility of parallel execution of alternative threads.

□

Figure 3.1 shows a simple process that uses all the five patterns using graphical notation that we have adopted for the remainder of this chapter.

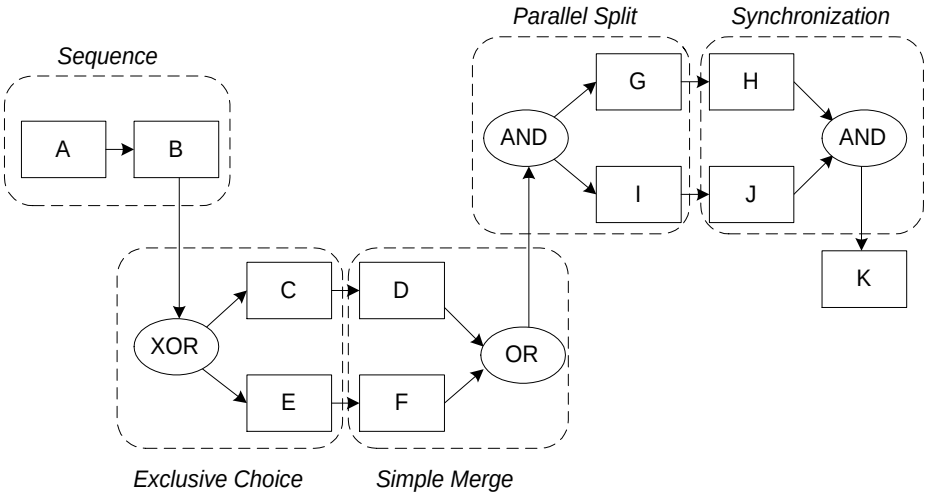

Figure 3.1: Graphical representation of basic control flow patterns

3.2 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 3.2.

□

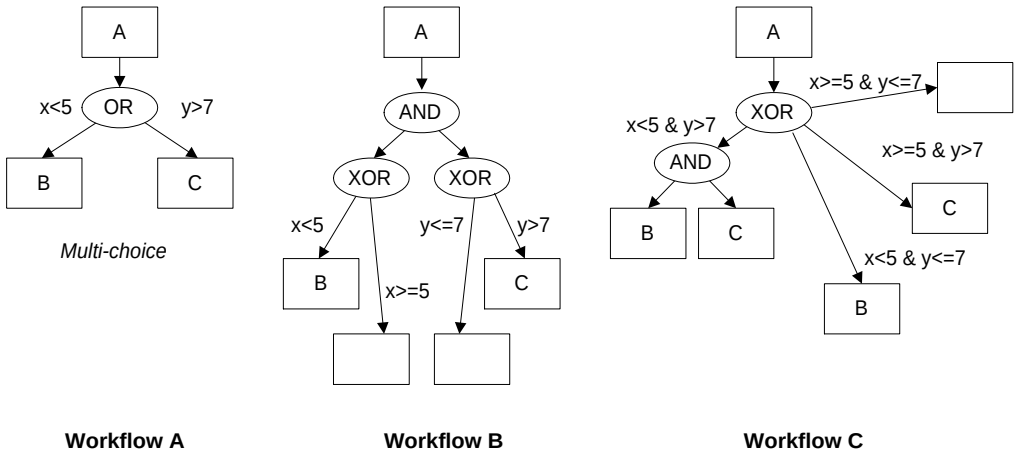

Figure 3.2: Implementation strategies for the Multi-Choice Pattern

It should be noted that there is a trade-off between implementing the multi-choice as in Workflow A of Figure 3.2 or as in Workflow B or 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 B and 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 3.3. After 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).

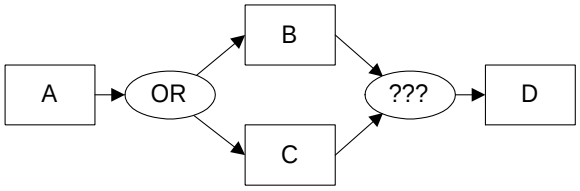

Figure 3.3: How do we want to merge here?

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 only workflow engine reviewed in this thesis that provide a straightforward construct for the realization of this pattern is MQSeries Workflow. 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.

- 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 3.4. 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*.

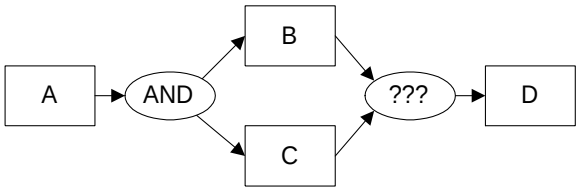

Figure 3.4: How do we want to merge here?

Pattern 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 3.4 due to syntactical restrictions that are imposed.
Implementation

- The merge constructs of 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 as presented on Figure 3.5. Note that we have decided not to use a separate graphical notation for the multi-merge construct. This should not cause any ambiguities as patterns Simple-Merge and

Multi-Merge which use the same, OR-Join graphical representation are used in different context.

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).

□

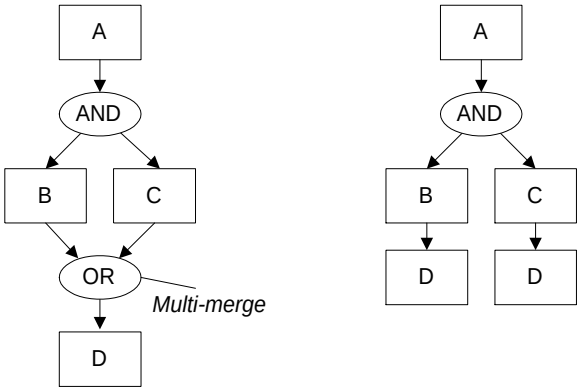

Figure 3.5: Typical implementation of Multi-Merge Pattern

Pattern 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 3.4 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. More specifically, as Changengine will not create a second instance of an activity, an OR-Join construct can be used to implement the Discriminator semantics.

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

- Typically, in other workflow engines the discriminator is impossible to implement directly in the workflow modelling 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 thesis. The discriminator needs to keep track of which branches have completed (and how often in case of multiple activations of the same branch) and reset 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 3.6.

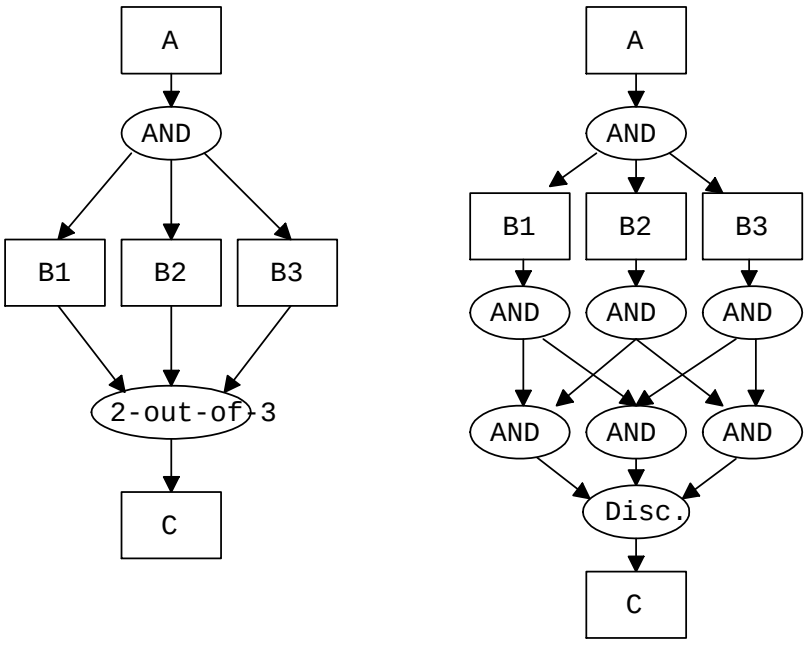

Figure 3.6: Implementation of a 2-out-of-3-Join using the basic discriminator

3.3 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.

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).

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) 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 either through auxiliary variables or through node repetition. A thorough analysis of conversions and an identification of some situations where they cannot be done is presented in Chapter 5 and Chapter 6.

□

Another example of a requirement imposed by some workflow products is that the workflow model has 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 (Staffware, MQSeries Workflow) 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 3.4) 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 presented in Section 6.1.

□

3.4 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 behaviour 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.

Problem Most workflow engines do not allow multiple instances of an activity to be active at the same time.

Implementation

- The most straightforward implementation of this pattern is through the use of the merge construct used in a loop in conjunction with the parallel split construct. This is possible in languages such as Forté and Verve. This solution is illustrated in Figure 3.7.

- 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*. Referring back to the process in Figure 3.7, one can use spawning-off facility instead of AND-Split to achieve a similar result

- Some workflow engines allow for creating an arbitrary number of multiple instances as long as this number is known at a certain point of a process execution. An example of such a facility is the *Bundle* in MQSeries Workflow and *Table-driven Dynamic Parallel Processing* in SAP R/3 Workflow. As these facilities provide an automatic synchronization of created instances, they are covered in detail when describing pattern 14 (Multiple Instances With a Priori Runtime Knowledge).

- For the workflow engines that lack a spawning-off facility, there is usually the possibility 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.

□

Pattern 12 is supported by many workflow management systems, typically through asynchronous subprocess invocation. 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.

Pattern 13 (Multiple Instances With a Priori Design Time Knowledge)

Description For one process instance an activity is enabled multiple times. The

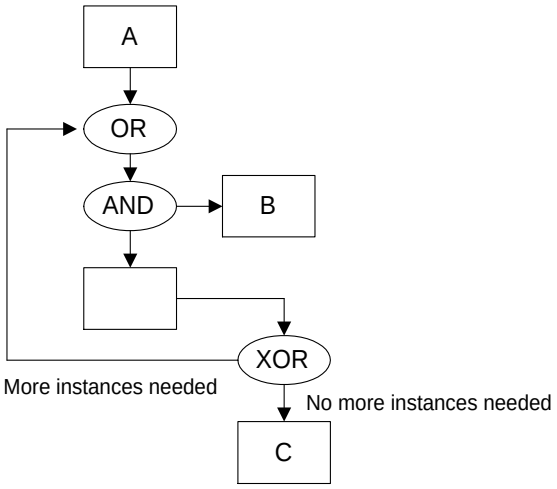

Figure 3.7: Implementation strategy for multiple instances

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. The workflow on the left diagram of Figure 3.8 is an implementation for the scenario when the number of instances is known to be less than four.

□

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.

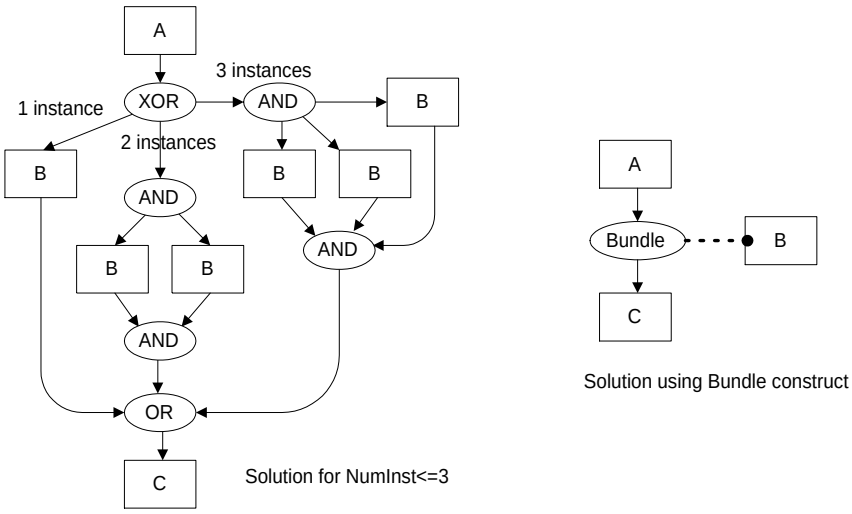

Figure 3.8: Design patterns for multiple instances

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

- As pattern 15 (Multiple Instances without a priori Runtime Knowledge) is a generalization of this pattern, any solution presented there can be used to implement this pattern.

- 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) and *Table-driven Dynamic Parallel Processing* in SAP R/3 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. Once all instances in a bundle are completed, the next activity is started (see right workflow in Figure 3.8).

□

Finally, we 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.

Pattern 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

- 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 sub-workflow containing the loop generating the multiple instances inside the decomposition block (see workflow in Figure 3.9). Here, activity B will be invoked many times. Once all instances of B are completed, the subprocess will complete and activity C 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.

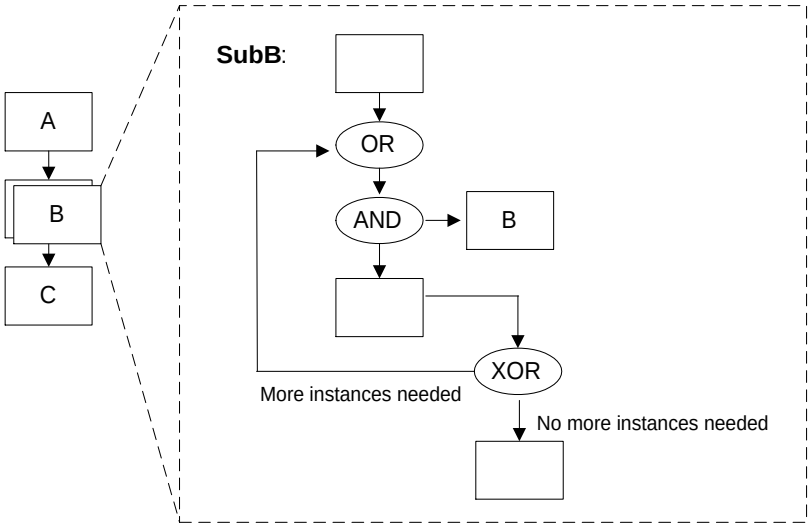

Figure 3.9: Implementation strategy for multiple instances

- If the workflow engine supports multiple instances directly (cf. Forté and Verve),

we can try and use the solution illustrated in Figure 3.7. However, activity C 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 modelling 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 C can be preceded by another activity that consumes the events from the queue and triggers C only if the number of events in the queue is equal to the number of instances of activity B. 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, 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.

□

3.5 State-based Patterns

In real workflows, most workflow instances are in a state awaiting processing rather than being processed. However, most workflow management systems abstract from states as they 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 that we are presenting in this section have all in common that a state of activity (typically the fact whether it is enabled or disabled) can be dynamically influenced by the state of another activity in a workflow process. As in most workflow management systems states are implicit, there are no convenient means to disable activities that have been once enabled (for example by sending negative messages). As we will see, these systems typically have problems dealing with the patterns introduced in this section.

The first pattern we would like to present can be seen as an extension to pattern 4 (Exclusive Choice). Moments of choice, such as supported by this pattern are of an *explicit* nature, i.e. it is based on data 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. There are situations when we would like both branches of the choice to be available for the end users (as if they were to be executed concurrently), however, once any of the scheduled activity from either of the branch is executed by one of the assignees,

the other activity should be disabled (and, in effect, disappear from the worklist of the assignee). We will call such a pattern a *Deferred Choice*.

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 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.

- 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

- 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 cancelled. Consider the deferred choice between B and C in Figure 3.10 (Workflow 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. Workflow B of Figure 3.10 shows the corresponding workflow model. Note that the solution does not always work because B and C can be selected/executed concurrently.

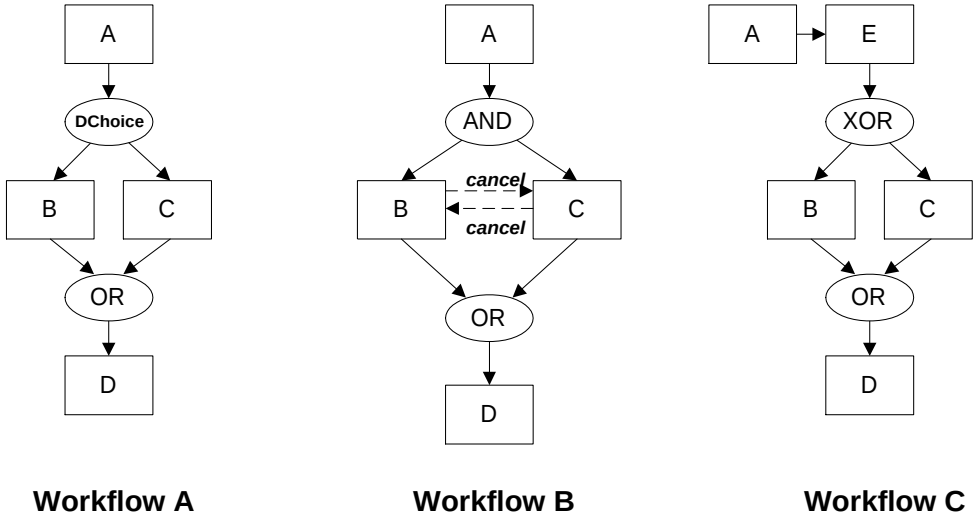

Figure 3.10: Strategies for implementation of deferred choice

- 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 external 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 3.10. By introducing a new activity E after A and redirecting external 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 C of Figure 3.10 shows the corresponding workflow model. Note that the solution moves part of the routing to the application or task level. Moreover, this solution assumes that the choice is made based on the type of external 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 3.11 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 3.11.

- 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".

□

The next pattern, Pattern 18 (Milestone), allows for testing whether an activity (or, more generally, a process instance) has reached a certain state. 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 an activity or a case is in a specific state.

Example 3.5.1 Consider the workflow process for handling complaints (see Figure 3.12). 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 Questionnaire*). 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 Questionnaire*), the complaint is processed or not. The actual processing of the complaint (activity *Process Complaint*), if it is to be made at all, 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* is called a milestone.

Pattern 18 (Milestone)
Description The enabling of an activity depends on some other activity(s) 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. We would like activity A to be enabled if and only if an activity B has been executed and

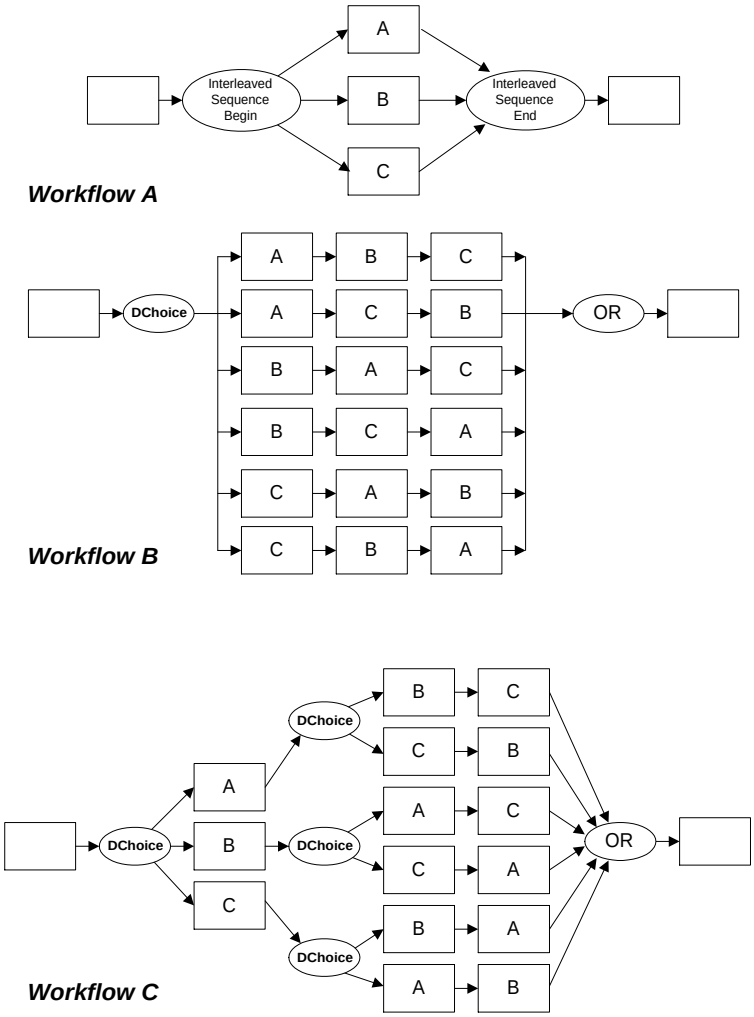

Figure 3.11: The implementation options for interleaving execution of A, B and C.

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 3.13 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 impact the state of the process. It only tests whether milestone is reached so that it can be executed.

Synonyms Test arc, deadline (cf. [JB96]), state condition, withdraw message.

Examples

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

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

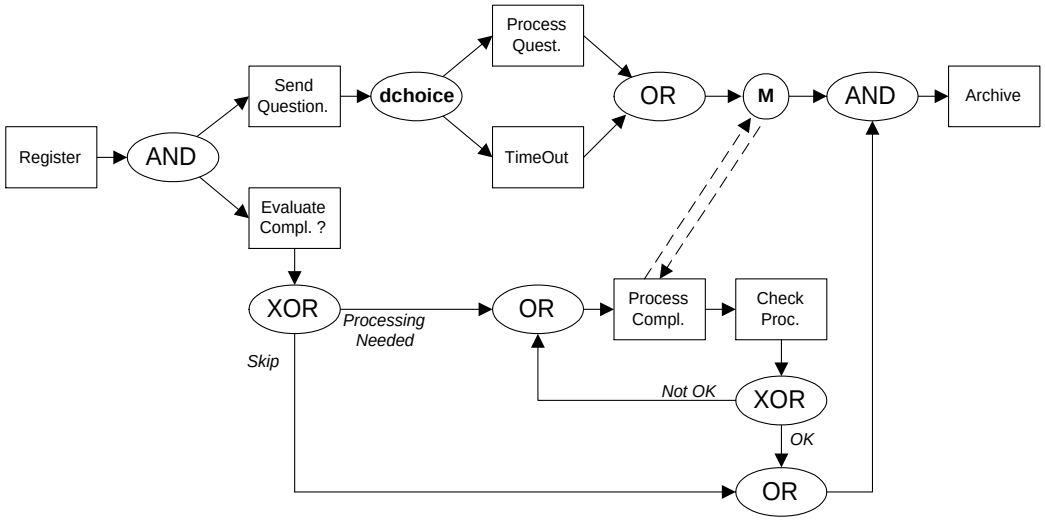

Figure 3.12: The state in-between the processing/time-out of the questionnaire and archiving the complaint is an example of a milestone.

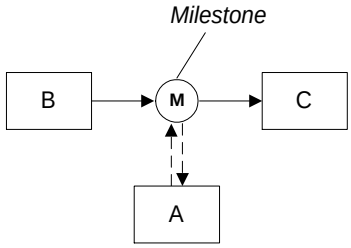

Figure 3.13: Schematical representation of a milestone.

delivery.

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

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 3.12 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

- The simple milestone scenario from Figure 3.13 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 A in Figure 3.14. 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 3.12 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 B in Figure 3.14. 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.

□

3.6 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

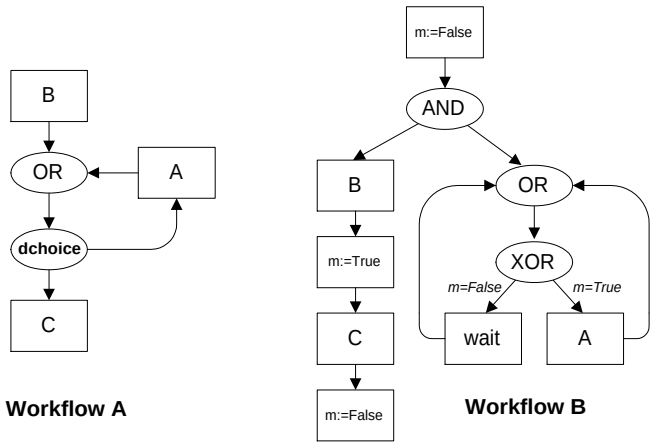

Figure 3.14: Implementation options for Milestone Pattern

and after the execution of activity C activity B is withdrawn. (See Figure 3.10: 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 modelling 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. Note that the drawback of this solution is the introduction of activities which do not correspond to actual steps of the process.

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

□

A similar construct is the cancellation of an entire case.

Pattern 20 (Cancel Case)
Description A case, i.e. workflow instance, is removed completely.
Synonyms Withdraw case.
Examples

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

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

Problem Workflow management systems typically do not support the withdrawal of an entire case using the (graphical) workflow language.
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, many workflow management systems support the withdrawal of cases using an API which simply removes the corresponding entries from the database.

□

3.7 Summary

The workflow patterns described in this Chapter correspond to routing constructs encountered when modelling and analysing 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.

The provision of the patterns would not be complete without a close look at to what extent current crop of workflow management systems is able to support them. In this thesis we have introduced the functionality of 8 workflow management systems. Tables 3.1 and 3.2 summarize the results of the comparison of these workflow management systems in terms of the selected patterns. For each product-pattern combination, we checked whether it is possible to realize the workflow pattern with the tool. As each pattern is different, it is hard to come up with a characterization that would fit all of them. 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. The exact evaluation of each pattern for every product evaluated with detailed comments is given in the Appendix A.

Pattern	**Product**			
	Staffware	MQSeries Workflow	Forté Conductor	Verve
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 3.1: The main results for Staffware, MQSeries Workflow, Forté Conductor and Verve.

From the comparison it is clear that no tools support all the selected patterns. In fact, many of these tools only support a fraction of these patterns and the best of them only support about 50%. Specifically the limited support for the discriminator, and its

Pattern	**Product**			
	Visual WorkFlo	Changengine	i-Flow	SAP R/3
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 3.2: The main results for Visual WorkFlo, Changengine, i-Flow, and SAP R/3 Workflow.

generalization, the N-out-of-M-join, the state-based patterns, the synchronization of multiple instances (no tool fully supports this) and cancellation (esp. of activities), is worth noting. Also, observe that Staffware is the only workflow management systems adopting a non-synchronizing strategy that support implicit termination.

It is our industry experience that very often workflow product vendors, when confronted with questions as to how certain complex patterns need to be implemented in their product, they respond that the workflow implementors 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 Chapter 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.

Chapter 4

Formal Foundations

So far we have attempted to characterise various workflow modelling techniques through the use of a very pragmatic approach, namely the workflow test harness and workflow patterns. Whilst these techniques have an important use, especially in industry where a quick evaluation of new products is needed, they do not provide us with a precise answer as to what is the *theoretical* expressiveness limit of a given modelling technique. In this Chapter we provide the necessary theoretical foundation to achieve the afore-mentioned goal.

As it turns out, workflow languages can, as far as the control flow perspective goes, be fully characterized in terms of the evaluation strategy they use, the concepts they support, and the syntactic restrictions they impose. Based upon the evaluation strategy, a mapping of workflows to Petri nets (see e.g. [Pet81, Rei85, Mur89, RR98]) is presented. Petri nets were chosen as they provide a general, well understood and well researched, theory for concurrency.

Petri nets have been proposed for modelling workflow process definitions long before the term "workflow management" was coined and workflow management systems became readily available. Consider for example the work on Information Control Nets, a variant of the classical Petri nets, in the late seventies [Ell79]. Petri nets constitute a good starting point for a solid theoretical foundation of workflow management. Clearly, a Petri net can be used to specify the control-flow, i.e., the routing of cases (workflow instances) [Aal98c]. Activities are modelled by transitions and causal dependencies are modelled by places, transitions, and arcs. In fact, a place corresponds to a condition which can be used as pre- and/or post-condition for activities. An AND-Split corresponds to a transition with two or more output places, and an AND-Join corresponds to a transition with two or more input places. OR-Splits/OR-Joins correspond to places with multiple outgoing/ingoing arcs.

The mappings presented, assigning a formal semantics to workflow languages, to-

gether with the "right" notion of equivalence, then allow an in-depth investigation into expressiveness properties of various classes of workflow languages presented in subsequent chapters.

4.1 Classification of Workflow Models

As we have seen in previous chapters, there seems to be not one established workflow modelling technique which is widely accepted in industry. All the products we have evaluated differ substantially in their interpretation of the basic workflow modelling constructs. Both the Test Harness presented in Section 2.3 and the support for specific patterns led us to classify workflow languages that we have considered, in terms of four evaluation strategies used. Specifically, only two products, Verve and Forté Conductor support Pattern 8 (Multi-merge) and they have very similar support for the rest of the patterns. We will classify them into the class of *Standard Workflows Models* with the main characteristic being the ability to create multiple concurrent instances of one activity. In contrast, Staffware, Changengine and i-Flow do not have support for Pattern 8 (Multi-Merge), otherwise they are very similar to Standard Workflows. We will classify them as *Safe Workflow Models*. Both SAP R/3 Workflow and FileNet Visual WorkFlo do not support Pattern 10 (Arbitrary Cycles) and through the Test Harness we have learned that due to syntactical restrictions certain processes cannot be modelled. We will consider these products to be members of a class called *Structured Workflow Models*. Finally, only one product, MQ Series Workflow, supports Pattern 7 (Synchronizing Merge) and we will consider it a member of a class which we will refer to as *Synchronizing Workflow Models*.

It should be stressed that these evaluation strategies may not take into account many semantical subtleties as well as more advanced, non-standard modelling constructs present in these products and as such should be viewed as an "idealization" of their approach. It is quite conceivable that products exist that escape this classification and undoubtedly such products may emerge in the future. The next subsections will provide a formal foundation for each of the four classes.

4.1.1 Standard Workflow Models

Standard Workflow Models represent what would appear to be the most "natural" interpretation of the WfMC definitions. In this section a mapping of the WfMC basic control flow constructs to Petri nets is provided, which captures this interpretation formally. In addition to the mapping a justification is supplied as to why we think this mapping represents the "intent" of the broader workflow community and a discussion of how it compares to some other mappings which have been proposed.

In the vast majority of workflow management systems when an activity instance is finished, the next activity instance to be executed is selected and its state is changed to `READY` (this typically corresponds to placing it on a designated worklist). After this, the activity instance can go through a number of internal states. Finally, if all the associated processing has been performed successfully, its state is changed to `COMPLETE`. As will be seen, these two states are crucial to control flow considerations and any formal semantics of control flow constructs has to take at least these two states into account explicitly.

When using Petri nets for capturing formal semantics of workflows, there is a choice of labelling places or transitions, where the labels represent activities that are to be performed. We have chosen to label transitions as this appears to be more common. In this approach, a labelled transition being enabled indicates that the corresponding activity is in the `READY` state. Firing the transition then corresponds to executing the activity and changing its state to `COMPLETE`.

Not all transitions are labelled. Transitions without a label (sometimes the label λ is used, representing an internal or "silent" action; in the rest of this thesis we will refer to such transitions as λ-transitions) represent internal processing performed by the workflow engine which cannot be observed by the external users (though they may also represent execution of the so-called null activity, the activity which does nothing). Such transitions will play an important role when considering workflow equivalence. With these assumptions in mind, Figure 4.1 shows the semantics of basic workflow constructs.

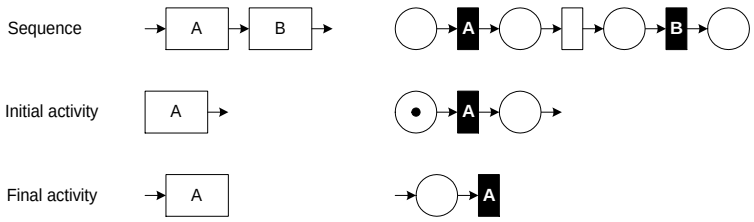

Figure 4.1: Mapping of basic control flow constructs

For the semantics of the XOR-Split, consider Figure 4.2. Two alternative mappings are shown with the rightmost mapping commonly used in the literature (see e.g. [AAH98, SH96]). The semantics of the XOR-Split is that when completing activity A, a choice needs to be made for activity B or activity C. However, only one of them can be in the state `READY`. Hence, the rightmost mapping is incorrect, as after completion of activity A both activities B and C would be enabled (though still only one of them will be actually executed). This is not what can be observed for the majority of workflow systems - either B or C is enabled (appears on the worklist) but not both.

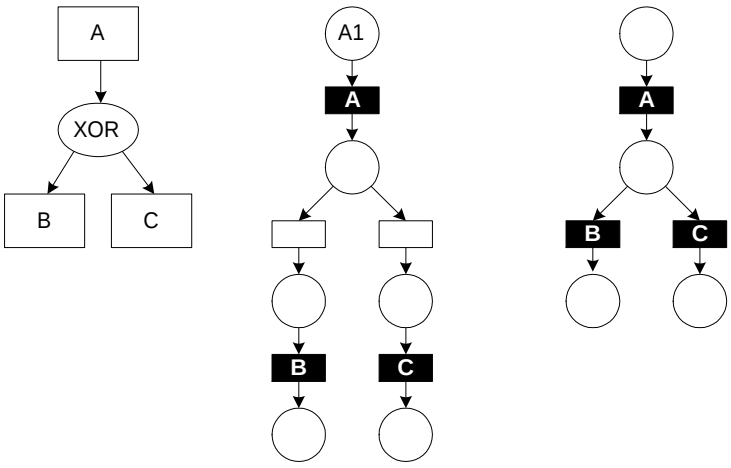

Figure 4.2: Alternative mappings for the XOR-Split

The rightmost mapping would correspond to the Deferred Choice pattern, introduced in section 3.5, and its importance will be immediate in later sections when discussing the expressiveness of Standard Workflow Models. Note that the semantics of the XOR-Split has been presented for the binary case, but, of course, can be trivially extended for the n-ary case (this will also hold for the other constructs presented in this Thesis).

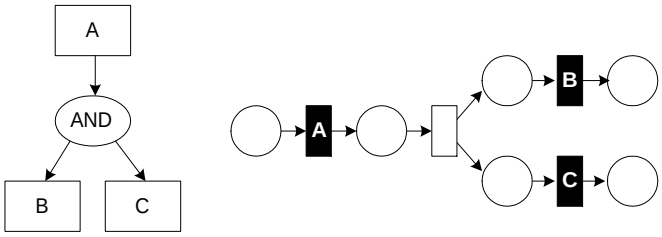

Figure 4.3: Mapping for the AND-Split

The interpretation of the AND-Split is presented in Figure 4.3, while interpretations for the AND-Join and the OR-Join are provided in Figure 4.4.

Note that the formal semantics provided for both types of joins leaves no ambiguities as to what is the semantics of these constructs when put in the context of a more complicated process structure. For example, if the AND-Join and activities A and B of Figure 4.4 were preceded by a XOR-Split, only one incoming activity (say, activity B) could complete. In this case there would be a token in c_B and the subsequent transition will never fire as no token would ever reach c_A. The net would then be in deadlock. If, on the other hand, the OR-Join and activities A and B of Figure 4.4 were preceded by the AND-Split, both activities B and A could run in parallel and

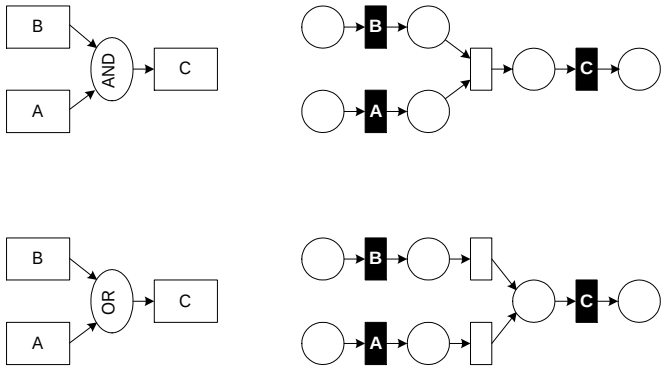

Figure 4.4: Mappings for the AND-Join and the OR-Join

tokens would be produced for both c_B and c_A. As a result two tokens would end up in r_C (though not necessarily at the same point in time). This corresponds to a situation where activity C has to be performed twice (which may or may not be desirable). In a workflow context this behaviour is observable, as any user that has been assigned to perform activity C will see two instances of this activity on his/her worklist.

Having informally established what a Standard Workflow Model is and how its constructs should be mapped to Petri nets, the formal definition of such a net (Definition 4.1.1) and its mapping (Definition 4.1.3) can be presented.

Definition 4.1.1

A Standard Workflow Model *is a tuple* $\mathcal{W} = \langle \mathcal{P}, \mathcal{J}_o, \mathcal{J}_a, \mathcal{S}_o, \mathcal{S}_a, \mathcal{A}, \mathsf{Trans}, \mathsf{Name} \rangle$ *where* $\mathcal{P}$ *is a set of process elements which can be further divided into disjoint sets of OR-Joins* $\mathcal{J}_o$*, AND-Joins* $\mathcal{J}_a$*, XOR-Splits* $\mathcal{S}_o$*, AND-Splits* $\mathcal{S}_a$*, and activities* $\mathcal{A}$*;* $\mathsf{Trans} \subseteq \mathcal{P} \times \mathcal{P}$ *is a transition relation between process elements and* $\mathsf{Name}: \mathcal{A} \rightarrow \mathcal{N}$ *is a function assigning names to activities taken from some given set of names* $\mathcal{N}$ *containing special label* λ*.*

Activities without names[1] *are referred to as null activities. Joins have an outdegree of at most one, while splits have an indegree of at most one. Activities have an indegree and outdegree of at most one. Finally, we will call activities with an indegree of zero* initial items *(*$\mathcal{I} \subseteq \mathcal{A}$*) and all process elements with an outdegree of zero* - final items *(*$\mathcal{F} \subseteq \mathcal{P}$*).* □

As a shorthand representation for $\mathcal{P}^{\mathcal{W}}$ (the set of process elements $\mathcal{P}$ of a Standard Workflow Model $\mathcal{W}$) we will use just $\mathcal{P}$ if the model $\mathcal{W}$ is obvious from the context. Additionally we will use the notation $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ whenever there is no need to distinguish between the different types of process elements.

[1]For an activity a without a name we have $\mathsf{Name}(a) = \lambda$

In Definition 4.1.1 we have imposed as few as possible syntactic restrictions. In this respect the following is worth noting:

- It may seem to be very restrictive to require that activities have an indegree and outdegree of at most one (and similar restrictions for the splits). This approach has been chosen to avoid possible ambiguities. For example, an activity with an indegree of two is sometimes interpreted as an AND-Join and sometimes as an OR-Join. It is trivial to map any language with such implicit semantics to our explicit notation.
- Most languages would require that the indegree of joins is at least one. Similarly they would require the outdegree of splits to be at least one. We have decided not to impose these restrictions as by not introducing these restrictions we can greatly simplify our definition as well as some further proofs without loosing any generality.

Definition 4.1.2

Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ *be a Standard Workflow Model and* $e \in \mathcal{P}$ *a process element of* $\mathcal{W}$. *Input elements of* e *are given by* $in(e) = \{x \in \mathcal{P} \mid x \ \mathsf{Trans}\ e\}$ *and output elements of* e *by* $out(e) = \{x \in \mathcal{P} \mid e \ \mathsf{Trans}\ x\}$. □

Definition 4.1.3

Given a Standard Workflow Model $\mathcal{W} = \langle \mathcal{P}, \mathcal{J}_o, \mathcal{J}_a, \mathcal{S}_o, \mathcal{S}_a, \mathcal{A}, \mathsf{Trans}, \mathsf{Name} \rangle$, *the corresponding labelled Petri net* $PN_{\mathcal{W}} = \langle P_{\mathcal{W}}, T_{\mathcal{W}}, F_{\mathcal{W}}, L_{\mathcal{W}} \rangle$ *is defined by:*

$$
\begin{array}{lll}
P_{\mathcal{W}} = & \{r_{x,y} \mid x \in \mathcal{P} \wedge y \in in(x)\} \cup & \#\text{“ready” places}\# \\
& \{c_{x,y} \mid x \in \mathcal{P} \wedge y \in out(x)\} \cup & \#\text{“completed” places}\# \\
& \{r_x \mid x \in \mathcal{I}\} & \#\text{“initial” places}\# \\
 & & \\
T_{\mathcal{W}} = & \{X_{x,y} \mid x \in \mathcal{S}_o \wedge y \in out(x)\} \cup & \#XOR\text{-}Split\# \\
& \{R_x \mid x \in \mathcal{S}_a\} \cup & \#AND\text{-}Split\# \\
& \{K_x \mid x \in \mathcal{J}_a\} \cup & \#AND\text{-}Join\# \\
& \{Q_{x,y} \mid x \in \mathcal{J}_o \wedge y \in in(x)\} \cup & \#OR\text{-}Join\# \\
& \{A_x \mid x \in \mathcal{A}\} \cup & \#activity\# \\
& \{L_{x,y} \mid x \ \mathsf{Trans}\ y\} & \#connecting\ trans.\# \\
 & & \\
L_{\mathcal{W}} = & \{(A_x, \mathsf{Name}(x)) \mid x \in \mathcal{A}\} \cup & \#activities\# \\
& \{(t, \lambda) \mid t \in T_{\mathcal{W}} \wedge \neg\exists_{x \in A}\,[t = A_x]\} & \#other\ trans\# \\
 & & \\
F_{\mathcal{W}} = & \{(r_x, A_x) \mid x \in \mathcal{I}\} \cup & \#initial\ places\#
\end{array}
$$

$$
\begin{array}{ll}
\{(r_{x,y}, A_x) \mid x \in \mathcal{A} \wedge y \in in(x)\} \cup & \\
\{(A_x, c_{x,y}) \mid x \in \mathcal{A} \wedge y \in out(x)\} \cup & \#activity\# \\
\{(r_{x,y}, K_x) \mid x \in \mathcal{J}_a \wedge y \in in(x)\} \cup & \\
\{(K_x, c_{x,y}) \mid x \in \mathcal{J}_a \wedge y \in out(x)\} \cup & \#AND\text{-}Join\# \\
\{(r_{x,y}, R_x) \mid x \in \mathcal{S}_a \wedge y \in in(x)\} \cup & \\
\{(R_x, c_{x,y}) \mid x \in \mathcal{S}_a \wedge y \in out(x)\} \cup & \#AND\text{-}Split\# \\
\{(r_{x,y}, Q_{x,y}) \mid x \in \mathcal{J}_o \wedge y \in in(x)\} \cup & \\
\{(Q_{x,z}, c_{x,y}) \mid x \in \mathcal{J}_o \wedge y \in out(x) \wedge z \in in(x)\} \cup & \#OR\text{-}Join\# \\
\{(r_{x,y}, X_{x,z}) \mid x \in \mathcal{S}_o \wedge y \in in(x) \wedge z \in out(x)\} \cup & \\
\{(X_{x,y}, c_{x,y}) \mid x \in \mathcal{S}_o \wedge y \in out(x)\} \cup & \#XOR\text{-}Split\# \\
\{(c_{x,y}, L_{x,y}) \mid x \ \mathsf{Trans}\ y\} \cup & \\
\{(L_{x,y}, r_{y,x}) \mid x \ \mathsf{Trans}\ y\} & \#connecting\#
\end{array}
$$

□

Definition 4.1.4

Given a Standard Workflow Model $\mathcal{W}$, the corresponding net system of $\mathcal{W}$ is a pair $(PN_{\mathcal{W}}, M_0)$ where $PN_{\mathcal{W}}$ is the corresponding net and M_0 is an initial marking that assigns a single token to each of the places in $\{r_x \mid x \in \mathcal{I}\}$. □

We will often refer to Petri nets resulting from the translation of Standard Workflow Models as *Standard Workflow Nets*.

Though the definition of a Standard Workflow Model may look complicated, it is constructed from a number of elementary building blocks, which can be isolated through Definition 4.1.5 if required.

Definition 4.1.5

Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ be a Standard Workflow Model and $PN_{\mathcal{W}} = \langle P_{\mathcal{W}}, T_{\mathcal{W}}, F_{\mathcal{W}}, L_{\mathcal{W}} \rangle$ its corresponding net. The associated net of a process element $e \in \mathcal{P}$, $PN^e_{\mathcal{W}} = \langle P^e_{\mathcal{W}}, T^e_{\mathcal{W}}, F^e_{\mathcal{W}}, L^e_{\mathcal{W}} \rangle$, is a subnet of $PN_{\mathcal{W}}$ and is defined by:

$$
\begin{array}{rcl}
P^e_{\mathcal{W}} & = & \begin{cases} \{r_{e,i} \mid i \in in(e)\} \cup \{c_{e,o} \mid o \in out(e)\} & \text{if } e \notin \mathcal{I} \\ \{r_e\} \cup \{c_{e,o} \mid o \in out(e)\} & \text{if } e \in \mathcal{I} \end{cases} \\
T^e_{\mathcal{W}} & = & \{t \in T_{\mathcal{W}} \mid {\bullet t} \subseteq P^e_{\mathcal{W}} \wedge t{\bullet} \subseteq P^e_{\mathcal{W}}\} \\
F^e_{\mathcal{W}} & = & F_{\mathcal{W}} \cap (P^e_{\mathcal{W}} \times T^e_{\mathcal{W}} \cup T^e_{\mathcal{W}} \times P^e_{\mathcal{W}}) \\
L^e_{\mathcal{W}} & = & L_{\mathcal{W}}[T^e_{\mathcal{W}}]
\end{array}
$$

□

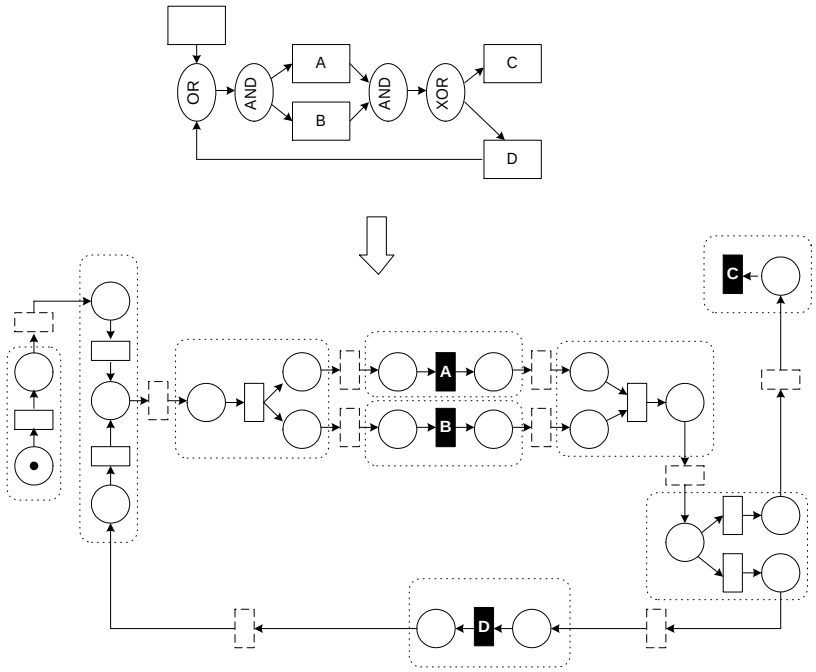

Figure 4.5: Sample Standard Workflow Model and its corresponding Petri net

Example 4.1.1 As an example of the application of Definitions 4.1.3 and 4.1.5 consider the Standard Workflow Model and its corresponding mapping along with associated nets in Figure 4.5. □

Having formally defined Standard Workflow Models, it is now possible to precisely define properties of such models, which have been informally referred to earlier in this chapter. First it is useful to be able to talk about running instances of workflows.

Definition 4.1.6
An instance *of a Standard Workflow Model $\mathcal{W}$ is a marking reachable from the initial marking of its corresponding net system.* □

The next definition formally defines what it means to *enable* a process element and *fire* a process element.

Definition 4.1.7
Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ be a Standard Workflow Model and $e \in \mathcal{P}$ a process element of $\mathcal{W}$. We say that e is enabled *in an instance M of $\mathcal{W}$ if any transition of its associated net $PN^e_{\mathcal{W}}$ is enabled in marking M. Similarly* firing *a process element e means firing any transition of its associated net.* □

Execution of a (finite) Standard Workflow Model leads either to a successful termination or to a deadlock or to an infinite loop from which the empty marking cannot be reached. More formally:

Definition 4.1.8

An instance of a Standard Workflow Model $\mathcal{W}$ is in deadlock *iff it is not the empty marking and no transition is enabled.* □

Definition 4.1.9

An instance of a Standard Workflow Model $\mathcal{W}$ is in an infinite loop *iff it is not the empty marking and there is no firing sequence that leads to either the empty marking or to a deadlock.* □

Definition 4.1.10

A Standard Workflow Model $\mathcal{W}$ is deadlock-free *iff none of its instances is in a deadlock.* □

Definition 4.1.11

A Standard Workflow Model $\mathcal{W}$ is terminating *iff from all its instances the empty marking can be reached.* □

Terminating Standard Workflow Models are similar to the class of WorkFlow Nets (WF-nets, cf. [Aal98c]). One of the differences is that WF-nets have an explicit termination place, i.e., a place which once it gets marked corresponds to a terminated workflow.

In previous chapters we have referred to the term *multiple instances* as the situation in which one activity may have many instances of it running concurrently. Based on our definition of the semantics of Standard Workflow Nets, we can provide a more formal definition.

Definition 4.1.12

A Standard Workflow Model does not have multiple instances *iff for every place p of its corresponding net system and for every reachable marking M, $M(p) \leq 1$. We will call such models* safe. □

Finally we will refer to Standard Workflows that are safe and terminating as *well-behaved.*

Definition 4.1.13

A Standard Workflow Model $\mathcal{W}$ is well-behaved *iff it is safe and it is terminating.*

□

4.1.2 Safe Workflow Models

The main difference between *Safe Workflow Models* and Standard Workflow Models is the behaviour of the OR-Join. As the WfMC does not define what should happen if more than one thread input to the OR-Join is concurrently active, some workflow management systems (e.g. Staffware, HP Changengine, Fujitsu's i-Flow) have been based on the assumption that subsequent active threads should never reach the OR-Join. Hence, their engines will never create multiple concurrent instances of the activity following the OR-Join. Though the actual solution is different for different products, from a conceptual point of view, the result is the same: there is no direct support for multiple instances.

To formally characterise such languages, two approaches could have been taken. One way would be to define the Petri net semantics of activities such that an activity's `READY` place can never hold more than one token. In that case it is guaranteed that the corresponding Petri net will be safe. This approach would try to formalize the *observable* behaviour of languages such as Staffware.

Following our discussions with vendors who have chosen the safe evaluation strategy, we have decided to take a different approach. It is based on the assumption that processes resulting in multiple active threads input to an OR-Join are considered to be flawed and their semantics is undefined.

This allows us to simply view Safe Workflow Models as a subclass of Standard Workflow Models.

Definition 4.1.14

A Safe Workflow Model *is a Standard Workflow Model such that its corresponding net system is safe.*

□

4.1.3 Structured Workflows Models

The philosophy behind the third class of languages, namely Structured Workflow Models, is that the workflow user should not be allowed to model processes that result in semantical ambiguities such as in situations where an AND-Split is followed by an OR-Join. To achieve that, they take a more stringent view on the allowable relations between different workflow model constructs.

In this approach the semantics of the individual workflow constructs is exactly the same as for Standard Workflow nets, however, there are a number of syntactical restrictions. Intuitively a structured workflow is a workflow where each XOR-Split

has a corresponding OR-Join and each AND-Split has a corresponding AND-Join and arbitrary cycles are not allowed.

Definition 4.1.15

1. *A Proper Structure is a Standard Workflow Model inductively defined as follows:*
   (a) *A single activity is a Proper Structure. This activity is both the* initial *and the* final *activity of this Proper Structure.*
   (b) *Let X and Y be Proper Structures. The concatenation of these workflows, where the final activity of X has a transition to the initial activity of Y, then also is a Proper Structure. The initial activity of this Proper Structure is the initial activity of X and its final activity is the final activity of Y.*
   (c) *Let $X_1, \ldots, X_n$ be Proper Structures and let J be an OR-Join and S a XOR-Split. The workflow with as initial activity S and final activity J and transitions between S and the initial activities of X_i, and between the final activities of X_i and J, is then also a Proper Structure. The initial activity of this Proper Structure is S and its final activity is J.*
   (d) *Let $X_1, \ldots, X_n$ be Proper Structures and let J be an AND-Join and S an AND-Split. The workflow with as initial activity S and final activity J and transitions between S and the initial activities of X_i, and between the final activities of X_i and J, is then also a Proper Structure . The initial activity of this Proper Structure is S and its final activity is J.*
   (e) *Let X and Y be Proper Structures and let J be an OR-Join and S a XOR-Split. The workflow with as initial activity J and as final activity S and with transitions between J and the initial activity of X, between the final activity of X and S, between S and the initial activity of Y, and between the final activity of Y and J, is then also a Proper Structure. If X is a null activity, then graphically it is not explicitly represented; an arrow is drawn directly from S to J. Similarly, if Y is a null activity then an arrow is drawn directly from J to S. The initial activity of this Proper Structure is J and its final activity is S.*
2. *A Proper Structure with an activity as the inital item and an activity as the final item is a Structured Workflow Model.*

□

The above definition is illustrated in Figure 4.6. Note that clause (e) of the definition would correspond to a classic WHILE-loop if X is a null activity and to a classic

REPEAT-UNTIL-loop if Y is a null activity. If $n = 2$, the clause (c) corresponds to a classic IF-THEN-ELSE.

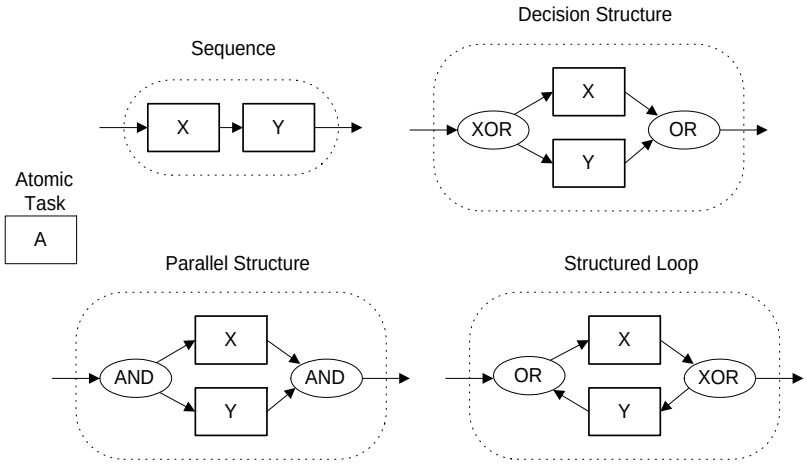

Figure 4.6: Illustration of Structured Workflow Models

As is clear from definition 4.1.15, every Structured Workflow Model is also a Standard Workflow Model. The reader may also note that every Structured Workflow Model will always have one initial and one final task.

Examples of commercial products that fit this category are FileNet's Visual WorkFlo and SAP R/3 Workflow. In both languages it is possible to design structured models only. These models resemble the definition provided earlier very closely with some minor exceptions such as that in Visual WorkFlo the loops can only be of the form "WHILE p DO X". In SAP R/3 Workflow the modeller has a choice between a "WHILE" loop and an "UNTIL" loop.

Example 4.1.2 Figure 4.7 presents an example of a Structured Workflow Model. □

4.1.4 Synchronizing Workflow Models

Synchronizing Workflow Models form a fourth class of workflow languages based on yet another, fundamentally different, interpretation of the WfMC definitions of the basic control flow constructs. The intuitive reasoning here is as follows. An AND-Join typically follows an AND-Split and can be seen as a construct that synchronizes a number of active threads. An OR-Join on the other hand, typically follows an exclusive XOR-Split. While there is only one active thread of execution in that case,

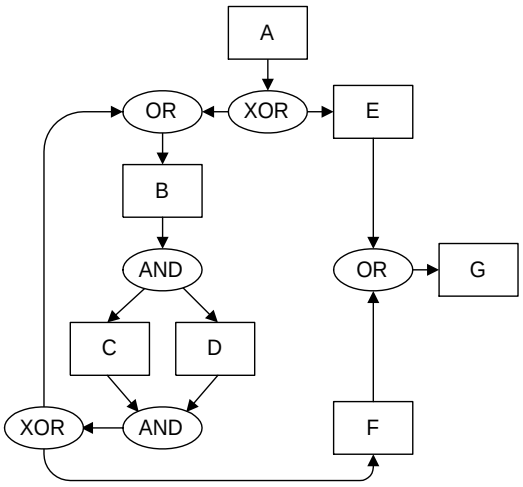

Figure 4.7: Example of a Structured Workflow Model

the OR-Join can still be seen as a construct that synchronizes threads: one active and the others inactive. The active thread propagates a "True" token, while an inactive thread propagates a "False" token. Hence both types of joins synchronize a number of threads of execution. With slight modifications, this view was successfully implemented by IBM's MQSeries Workflow (formerly known as FlowMark). Note that a synchronizing strategy prevents the use of arbitrary cycles (as that would immediately lead to a deadlock). Later it will be proven that Synchronizing Workflow Models never deadlock.

The semantics of Synchronizing Workflow Models is very naturally captured using Coloured Petri nets [Jen87] or Predicate/Transition nets [Gen87], as they allow for typed tokens with identity. However, in order to facilitate a formal comparison with Standard Workflow Models, we provide a formal semantics in terms of standard Petri nets.

In Synchronizing Workflow Models, an activity can receive two types of tokens, a true token or a false token. Receipt of a true token should enable the activity, while receipt of a false token should lead to the activity being skipped and the token to be propagated. To capture this in standard Petri nets, we divide the set of places into places that capture the receipt of true tokens ("true places") and places that capture the receipt of false tokens ("false places"). This leads to the semantics represented in Figure 4.8.

Each activity x has a place $rt_{x,y}$ in its corresponding Petri net (where y corresponds to the input element of x, if existing) that will hold a token if the activity received a true token from y, and a place $rf_{x,y}$ that will hold a token if the activity received a false token from y. As is clear from the net in Figure 4.8, a token in a "true" place

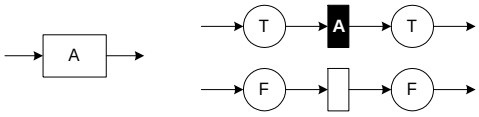

Figure 4.8: Activity semantics for Synchronizing Workflow Models

will lead to the transition labelled with A being enabled, while a token in its "false" place will lead to the non-labelled transition being enabled, and hence nothing, other than propagation of the token, will happen.

The semantics of the XOR-Split and the AND-Split is relatively straightforward. When a true token arrives, a XOR-Split will pass on a true token to one of its outgoing branches and false tokens for all the other outgoing branches. When a true token arrives for an AND-Split, true tokens are passed on to all its outgoing branches. Both splits behave similar when receiving a false token; it is simply passed on to all outgoing branches. This semantics is captured in Figure 4.9.

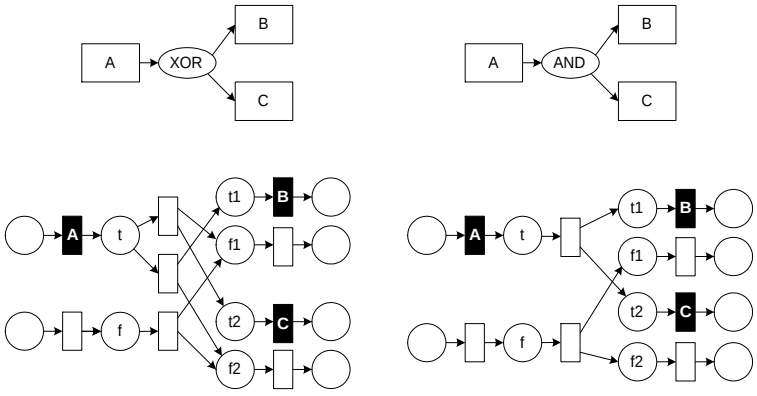

Figure 4.9: Split semantics for Synchronizing Workflow Models

More interesting is the semantics of the join constructs. As noted earlier, in Synchronizing Workflow Models a join construct always waits for a token to arrive from every incoming transition. The only differentiator between different types of joins could be the type of tokens expected. In this thesis we will follow MQSeries/Workflow in that we will distinguish two cases - an *ANY-Join* which passes on a true token if it received at least one true token (otherwise it passes on a false token) and the *ALL-Join* which passes on a true token if it received true tokens from all incoming branches (otherwise it passes on a false token). Later, in Section 5.4, we will show how the Synchronizing Workflow's *ANY-Join* and *ALL-Join* correspond to the Standard and Safe Workflow's *OR-Join* and *AND-Join*.

In Figure 4.10, the semantics of the joins is shown in the context of Synchronizing Workflow Models.

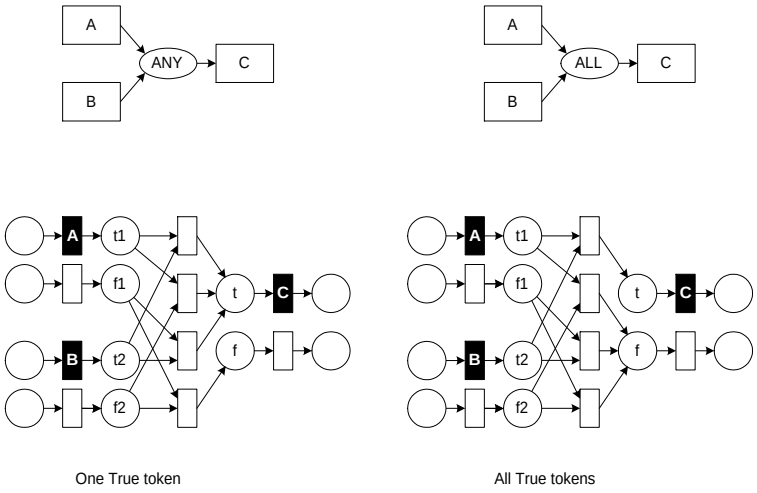

Figure 4.10: Join semantics for Synchronizing Workflow Models

Free-choice Petri net is a net in which the choice between two enabled transitions is never influenced by the rest of the system. On a structural level it means that Petri net is free-choice iff for every place and transition of this net if there is an arc from a place p to a transition t, then there must be an arc from any input place of t to any output transition of s (see Appendix B for a formal definition of a free-choice Petri net). One important characterisation of Synchronizing Workflow Models is that the Petri net representation of the join constructs is not free-choice (see [DE95] for a detailed discussion of free-choice Petri nets). In Section 5.4 it will be shown that some Synchronizing Workflow Models are inherently non free-choice.

Having informally established the semantics of Synchronizing Workflow Models, Definition 4.1.16 formally defines their syntax, while Definition 4.1.18 formally defines their semantics.

Definition 4.1.16

A Synchronizing Workflow Model *is a tuple* $\mathcal{W} = \langle \mathcal{P}, \mathcal{J}_o, \mathcal{J}_a, \mathcal{S}_o, \mathcal{S}_a, \mathcal{A}, \mathsf{Trans}, \mathsf{Name} \rangle$ *where* $\mathcal{P}$ *is a set of process elements which can be further divided into disjoint sets of ANY-Joins* $\mathcal{J}_o$*, ALL-Joins* $\mathcal{J}_a$*, XOR-Splits* $\mathcal{S}_o$*, AND-Splits* $\mathcal{S}_a$*, and activities* $\mathcal{A}$*;* $\mathsf{Trans} \subseteq \mathcal{P} \times \mathcal{P}$ *is a transition relation between process elements and* $\mathsf{Name}: \mathcal{A} \rightarrow \mathcal{N}$ *is a partial function assigning names to activities taken from some given set of names* $\mathcal{N}$ *containing special label* λ*.*

Activities without names are referred to as null activities. Joins have an indegree of at least one and an outdegree of one, while splits have an indegree of one and an outdegree of at least one. Activities have an indegree and outdegree of at most one. Finally, we will call activities with an indegree of zero initial items ($\mathcal{I} \subseteq \mathcal{A}$) and conversely, activities with an outdegree of zero - final items ($\mathcal{F} \subseteq \mathcal{A}$).

□

Note that the syntax of Synchronizing Workflow Models is very similar to the syntax of Standard Workflow Models. The only difference is that joins and splits cannot have indegree or outdegree of zero (this is to allow simplification of the semantics).

The following definition provides auxiliary functions and predicates that facilitate the specification of the formal semantics.

Definition 4.1.17

Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ be a Synchronizing Workflow Model and $p \in \mathcal{P}$ a process element. The input elements of p are given by $in(p) = \{x \in \mathcal{P} \mid x \ \mathsf{Trans}\ p\}$ and output elements of p by $out(p) = \{x \in \mathcal{P} \mid p \ \mathsf{Trans}\ x\}$. Further, if $b \in \{t, f\}^A$ is a function with domain A (which is nonempty), then $alltrue(b)$ holds iff $\forall_{a \in A}\,[b(a) = t]$ and $allfalse(b)$ holds iff $\forall_{a \in A}\,[b(a) = f]$. □

Definition 4.1.18

Given a Synchronising Workflow Model $\mathcal{W}$, the corresponding labelled Petri net $PN_{\mathcal{W}} = (P_{\mathcal{W}}, T_{\mathcal{W}}, F_{\mathcal{W}}, L_{\mathcal{W}})$ is defined by:

$$
\begin{array}{lll}
P_{\mathcal{W}} = & \{rt_{x,i} \mid x \in \mathcal{P} \wedge i \in in(x)\} \cup & \#\text{“ready” } true\# \\
& \{rf_{x,i} \mid x \in \mathcal{P} \wedge i \in in(x)\} \cup & \#\text{“ready” } false\# \\
& \{ct_{x,o} \mid x \in \mathcal{P} \wedge o \in out(x)\} \cup & \#\text{“completed” } true\# \\
& \{cf_{x,o} \mid x \in \mathcal{P} \wedge o \in out(x)\} \cup & \#\text{“completed” } false\# \\
& \{rt_x \mid x \in \mathcal{I}\} \cup & \#\text{“initial” } true\# \\
& \{rf_x \mid x \in \mathcal{I}\} & \#\text{“initial” } false\# \\
\\
T_{\mathcal{W}} = & \{XT_{x,o} \mid x \in \mathcal{S}_o \wedge o \in out(x)\} \cup \{XF_x \mid x \in \mathcal{S}_o\} \cup & \#XOR\text{-}Split\# \\
& \{RF_x \mid x \in \mathcal{S}_a\} \cup \{RT_x \mid x \in \mathcal{S}_a\} \cup & \#AND\text{-}Split\# \\
& \{K_x^b \mid x \in \mathcal{J}_a \wedge b \in \{t, f\}^{in(x)}\} \cup & \#ALL\text{-}Join\# \\
& \{Q_x^b \mid x \in \mathcal{J}_o \wedge b \in \{t, f\}^{in(x)}\} \cup & \#ANY\text{-}Join\# \\
& \{AF_x \mid x \in \mathcal{A}\} \cup \{AT_x \mid x \in \mathcal{A}\} \cup & \#activity\# \\
& \{LT_{x,y} \mid x \ \mathsf{Trans}\ y\} \cup \{LF_{x,y} \mid x \ \mathsf{Trans}\ y\} & \#connecting\ trans.\# \\
\\
L_{\mathcal{W}} = & \{(AT_x, \mathsf{Name}(x)) \mid x \in \mathcal{A}\} \cup & \#activities\# \\
& \{(t, \lambda) \mid t \in T_{\mathcal{W}} \wedge \neg\exists_{x \in A}\,[t = AT_x]\} & \#other\ trans\#
\end{array}
$$

$$
\begin{aligned}
F_{\mathcal{W}} = {} & \{(rt_x, AT_x) \mid x \in \mathcal{I}\} \cup \\
& \{(rf_x, AF_x) \mid x \in \mathcal{I}\} \cup && \#\textit{initial places}\# \\
& \{(rt_{x,i}, AT_x) \mid x \in \mathcal{A} \wedge i \in in(x)\} \cup \\
& \{(AT_x, ct_{x,o}) \mid x \in \mathcal{A} \wedge o \in out(x)\} \cup \\
& \{(rf_{x,i}, AF_x) \mid x \in \mathcal{A} \wedge i \in in(x)\} \cup \\
& \{(AF_x, cf_{x,o}) \mid x \in \mathcal{A} \wedge o \in out(x)\} \cup && \#\textit{activity}\# \\
& \{(rt_{x,i}, RT_x) \mid x \in \mathcal{S}_a \wedge i \in in(x)\} \cup \\
& \{(RT_x, ct_{x,o}) \mid x \in \mathcal{S}_a \wedge o \in out(x)\} \cup \\
& \{(rf_{x,i}, RF_x) \mid x \in \mathcal{S}_a \wedge i \in in(x)\} \cup \\
& \{(RF_x, cf_{x,o}) \mid x \in \mathcal{S}_a \wedge o \in out(x)\} \cup && \#\textit{AND-Split}\# \\
& \{(rf_{x,i}, XF_x) \mid x \in \mathcal{S}_o \wedge i \in in(x)\} \cup \\
& \{(XF_x, cf_{x,o}) \mid x \in \mathcal{S}_o \wedge o \in out(x)\} \cup \\
& \{(rt_{x,i}, XT_{x,o}) \mid x \in \mathcal{S}_o \wedge i \in in(x) \wedge o \in out(x)\} \cup \\
& \{(XT_{x,o1}, cf_{x,o2}) \mid x \in \mathcal{S}_o \wedge \{o1, o2\} \subseteq out(x) \wedge o1 \neq o2\} \cup \\
& \{(XT_{x,o1}, ct_{x,o1}) \mid x \in \mathcal{S}_o \wedge o1 \in outx\} \cup && \#\textit{XOR-Split}\# \\
& \{(rt_{x,i}, K^b_x) \mid x \in \mathcal{J}_a \wedge i \in in(x) \wedge b(i) = t\} \cup \\
& \{(K^b_x, ct_{x,o}) \mid x \in \mathcal{J}_a \wedge o \in out(x) \wedge alltrue(b)\} \cup \\
& \{(rf_{x,i}, K^b_x) \mid x \in \mathcal{J}_a \wedge i \in in(x) \wedge b(i) = f\} \cup \\
& \{(K^b_x, cf_{x,o}) \mid x \in \mathcal{J}_a \wedge o \in out(x) \wedge \neg alltrue(b)\} \cup && \#\textit{ALL-Join}\# \\
& \{(rt_{x,i}, Q^b_x) \mid x \in \mathcal{J}_o \wedge i \in in(x) \wedge b(i) = t\} \cup \\
& \{(Q^b_x, ct_{x,o}) \mid x \in \mathcal{J}_o \wedge o \in out(x) \wedge \neg allfalse(b)\} \cup \\
& \{(rf_{x,i}, Q^b_x) \mid x \in \mathcal{J}_o \wedge i \in in(x) \wedge b(i) = f\} \cup \\
& \{(Q^b_x, cf_{x,o}) \mid x \in \mathcal{J}_o \wedge o \in out(x) \wedge allfalse(b)\} \cup && \#\textit{ANY-Join}\# \\
& \{(ct_{x,y}, LT_{x,y}) \mid x \ \mathsf{Trans}\ y\} \cup \\
& \{(LT_{x,y}, rt_{y,x}) \mid x \ \mathsf{Trans}\ y\} \cup \\
& \{(cf_{x,y}, LF_{x,y}) \mid x \ \mathsf{Trans}\ y\} \cup \\
& \{(LF_{x,y}, rf_{y,x}) \mid x \ \mathsf{Trans}\ y\} && \#\textit{connecting ready/completed}\#
\end{aligned}
$$

□

Definition 4.1.19

Given a Synchronizing Workflow Model $\mathcal{W}$, the corresponding net system of $\mathcal{W}$ is a pair $(PN_{\mathcal{W}}, M_0)$ where $PN_{\mathcal{W}}$ is the corresponding net of $\mathcal{W}$ and M_0 is an initial marking that assigns a single token to each of the places in $\{rt_x \mid x \in \mathcal{I}\}$.

□

We will often refer to Petri nets resulting from the translation of Synchronizing Workflow Models as *Synchronizing Workflow Nets*.

Example 4.1.3 As an example of the application of Definition 4.1.18 consider the Synchronizing Workflow Model and its corresponding mapping in Figure 4.11. □

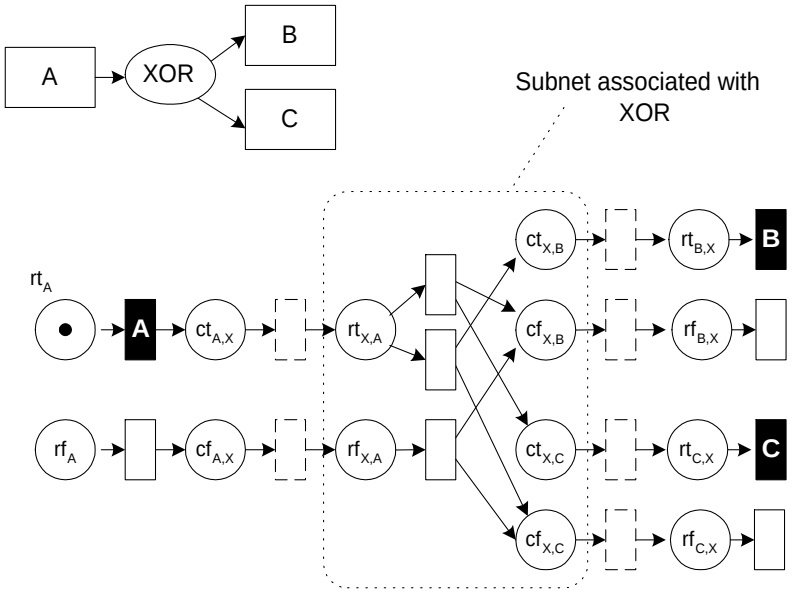

Figure 4.11: Synchronizing Workflow Model and its corresponding Petri net

Similarly to Standard Workflow Models, Synchronizing Workflow Models are constructed from a number of elementary building blocks, which can be isolated through Definition 4.1.20.

Definition 4.1.20

Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ be a Synchronizing Workflow Model and $PN_{\mathcal{W}} = \langle P_{\mathcal{W}}, T_{\mathcal{W}}, F_{\mathcal{W}}, L_{\mathcal{W}} \rangle$ its corresponding Petri net. Let $e \in \mathcal{P}_{\mathcal{W}}$ be a process element. The associated net, $PN^e_{\mathcal{W}} = \langle P^e_{\mathcal{W}}, T^e_{\mathcal{W}}, F^e_{\mathcal{W}}, L^e_{\mathcal{W}} \rangle$, a subnet of $PN_{\mathcal{W}}$, is defined by:

$$
\begin{array}{rcl}
P^e_{\mathcal{W}} & = & \begin{cases}
\{rt_{e,i} \mid i \in in(e)\} \cup \{rf_{e,i} \mid i \in in(e)\} \cup & \\
\{ct_{e,o} \mid o \in out(e)\} \cup \{cf_{e,o} \mid o \in out(e)\} & \text{if } e \notin \mathcal{I} \\
\{rt_e, rf_e\} \cup \{ct_{e,o} \mid o \in out(e)\} \cup \{cf_{e,o} \mid o \in out(e)\} \cup & \text{if } e \in \mathcal{I}
\end{cases} \\
T^e_{\mathcal{W}} & = & \{t \in T_{\mathcal{W}} \mid {\bullet t} \subseteq P^e_{\mathcal{W}} \wedge t{\bullet} \subseteq P^e_{\mathcal{W}}\} \\
F^e_{\mathcal{W}} & = & F_{\mathcal{W}} \cap (P^e_{\mathcal{W}} \times T^e_{\mathcal{W}} \cup T^e_{\mathcal{W}} \times P^e_{\mathcal{W}}) \\
L^e_{\mathcal{W}} & = & L_{\mathcal{W}}[T^e_{\mathcal{W}}]
\end{array}
$$

□

Example 4.1.4 The associated net for the XOR-Split is illustrated in Figure 4.11. □

Synchronizing Workflow Models have a more complicated Petri net translation because each process element can receive a "true" or a "false" token, and for that reason we introduce two input and two output places for each incoming and outgoing transition respectively. We will refer to places that can receive "true" tokens as "true" places of the net and places that can receive "false" tokens as "false" places of the net. This is captured formally in the following definition.

Definition 4.1.21

Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ *be a Synchronizing Workflow Model and* $PN_{\mathcal{W}}$ *its corresponding Petri net. The set of its* true places *is defined by*

$$\mathsf{True}^{\mathcal{W}} = \{rt_{x,i} \mid x \in \mathcal{P} \wedge i \in in(x)\} \cup \{ct_{x,o} \mid x \in \mathcal{P} \wedge o \in out(x)\} \cup \{rt_i \mid i \in \mathcal{I}\},$$

while the set of its false places *is given by:*

$$\mathsf{False}^{\mathcal{W}} = \{rf_{x,i} \mid x \in \mathcal{P} \wedge i \in in(x)\} \cup \{cf_{x,o} \mid x \in \mathcal{P} \wedge o \in out(x)\} \cup \{rf_i \mid i \in \mathcal{I}\}.$$

□

In an informal discussion earlier in this section we have often referred to the propagation of a "true" token or a "false" token. Formally we will call any token in a "true" place a "true" token and any token in a "false" place a "false" token.

Each incoming and outgoing transition of a process element has exactly one "true" place and one "false" place. The following definition captures the relationship between true and false places of the same workflow construct:

Definition 4.1.22

Let $\mathcal{W}$ *be a Synchronizing Workflow Model and* $PN_{\mathcal{W}}$ *its corresponding net. If* p *is a true place in the net* $PN_{\mathcal{W}}$*, then its corresponding false place* $\overline{p}$ *is* $rf_{x,y}$ *if* $p = rt_{x,y}$*,* rf_x *if* $p = rt_x$ *and it is* $cf_{x,y}$ *if* $p = ct_{x,y}$*. Similarly,* $\overline{p}$ *will yield the corresponding true place if* p *is a false place.* □

The definition of a workflow instance, deadlock and termination for Synchronizing Workflow Models are analogous to that of Standard Workflow Models. However, given the different Petri net translation the notion of a process element being *enabled* is slightly different and informally it means that for each incoming branch exactly one of the two (true or false) corresponding input places holds a token.

Definition 4.1.23

Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ *be a Synchronizing Workflow Model. A process element* $e \in \mathcal{P}$ *is* enabled *in a marking* M *of its associated net* $PN^e_{\mathcal{W}}$ *iff for all* x *such that* $x \in in(e)$

$$(M(rt_{e,x}) = 1 \wedge M(rf_{e,x}) = 0) \vee (M(rt_{e,x}) = 0 \wedge M(rf_{e,x}) = 1),$$

and for all y such that $y \in out(e)$

$$M(ct_{e,y}) = 0 \wedge M(cf_{e,y}) = 0.$$

□

In the context of Synchronizing Workflow Models it is also useful to talk about a process element being completed, which then means that for each outgoing branch exactly one of the two (true or false) corresponding output places holds a token.

Definition 4.1.24
Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ be a Synchronizing Workflow Model. A process element $e \in \mathcal{P}$ is completed *in a marking M of its associated net $PN^e_{\mathcal{W}}$ iff for all x such that $x \in in(e)$*

$$M(rt_{e,x}) = 0 \wedge M(rf_{e,x}) = 0,$$

and for all y such that $y \in out(e)$

$$(M(ct_{e,y}) = 0 \wedge M(cf_{e,y}) = 1) \vee (M(ct_{e,y}) = 1 \wedge M(cf_{e,y}) = 0).$$

□

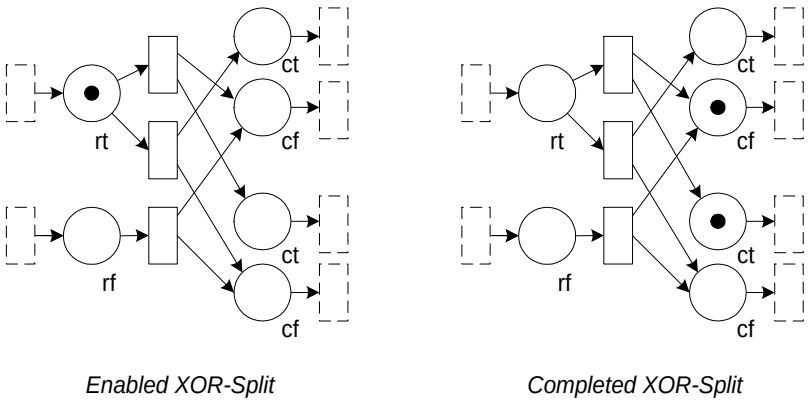

Figure 4.12: Enabled and completed XOR-Split

Example 4.1.5 Figure 4.12 shows markings in which an XOR-Split is *enabled* and *completed*. □

Finally, the following definition defines what it means to fire a process element.

Definition 4.1.25

Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ *be a Synchronizing Workflow Model and* $e \in \mathcal{P}$ *a process element which is enabled in marking* M *of its associated net* $PN^e_{\mathcal{W}}$. Firing e *means firing an enabled transition* t *of* $PN^e_{\mathcal{W}}$. □

The careful reader may notice that the definition of enabled process element for a Synchronizing Workflow Model is more restrictive than the similar definition for a Standard Workflow Model. This is due to the fact that the execution model for both workflows is fundamentally different. This issue will be further explored in Section 5.4.

4.2 Equivalence in the Context of Control Flow

Often workflow designers are faced with the task of transforming workflow specifications, for example to meet the particular requirements of a specific workflow engine. Naturally, such transformations should not alter the semantics of the original workflow, and as such they should be *equivalence preserving*. Similarly, when assessing the expressive power of a given workflow language the issue of equivalence is crucial. If one would like to prove that for a certain workflow a corresponding workflow in another language does, or does not, exist, this all depends on the notion of equivalence chosen.

For processes many different equivalence notions exist (e.g. trace, readiness, possible futures, fully concurrent bisimulation etc.). In fact, a whole area of research is devoted to this topic, referred to as comparative concurrency semantics (for an overview of many equivalence notions, refer to e.g. [Gla90, Gla93, PRS92]).

In the context of workflows, the choice of the "right" notion of equivalence is very much an open issue. The equivalence notion chosen should not be too restrictive as that would mean that workflows that one would like to consider as behaving identically, would be considered to be fundamentally different. Similarly, an equivalence notion should not be too relaxed, as it would identify workflows that behave fundamentally differently. Naturally this issue, to some extent, is open for debate as it depends on intuition as regards workflow execution and on what one considers to be a "workable" enough definition.

Consider the workflows $A1$ and $A2$ in Figure 4.13. These workflows produce identical traces, namely ab and ac. In other words they are *trace equivalent*. From a practical point of view, however, one would not like to consider them to be equivalent, as the moment of choice in both workflows is different. The choice for activity B or activity C may be influenced by the data produced by activity A in the left workflow, but not in the right. Clearly *trace equivalence* is not strong enough to distinguish these two

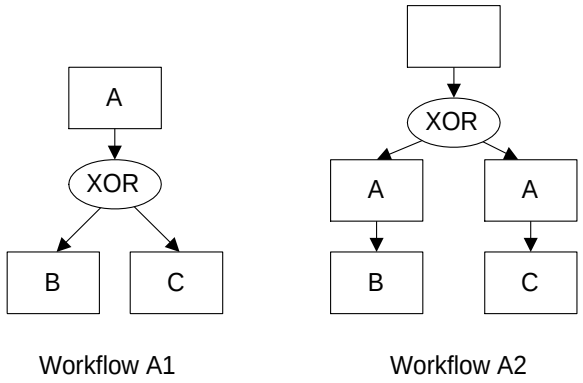

Figure 4.13: Two trace equivalent processes

workflows. Equivalence notions that take into account decision points are typically referred to as equivalence notions preserving *branching time* (as opposed to linear time). There are many equivalence notions that satisfy this criterion.

Considering only equivalence notions preserving branching time, we face a choice between *interleaving* semantics and the more complex *concurrent* semantics. In interleaving semantics, a process consisting of two tasks, A and B which run in parallel is equivalent to a process that chooses between running sequentially A followed by B or B followed by A. In other words, there is no *true concurrency*. As an example consider the two Petri nets, PN_1 and PN_2 of Figure 4.14. These two nets are equivalent under any interleaving equivalence notion. However we would like to consider workflows $B1$ and $B2$ of this Figure to be semantically different. The standard way of dealing with this problem (see e.g [BW90]) is to split a task into two observable transitions. Firing the first transition indicates *starting* of the task and firing the second transition indicates *completion* of the task. Having two parallel tasks A and B it is possible to obtain a trace $A_S B_S A_F B_F$ where A_S and B_S indicate start of tasks A and B respectively and A_F and B_F completion of tasks A and B respectively. This mapping is shown in Figure 4.14. Clearly workflows $B1$ and $B2$, given the presented mapping to Petri nets are not even trace equivalent. As the mappings to Petri nets, as presented in Sections 4.1.1 and 4.1.4 map tasks to a subnet containing one labelled transition, for two workflows to be equivalent we will require that (a) their corresponding Petri nets be equivalent and (b) the *begin-end* refinements of these Petri nets be equivalent where the begin-end refinement is a refinement that replaces every labelled transition with two labelled transitions and a place that is an output to the first transition and an input to the second transition (as in Figure 4.14).

Next consider Workflows $C1$ and $C2$ of Figure 4.15. Careful analysis of control flow taking into consideration the conditions of each of the XOR-Splits may lead to the conclusion that these workflows are equivalent. However, the Petri net mapping does

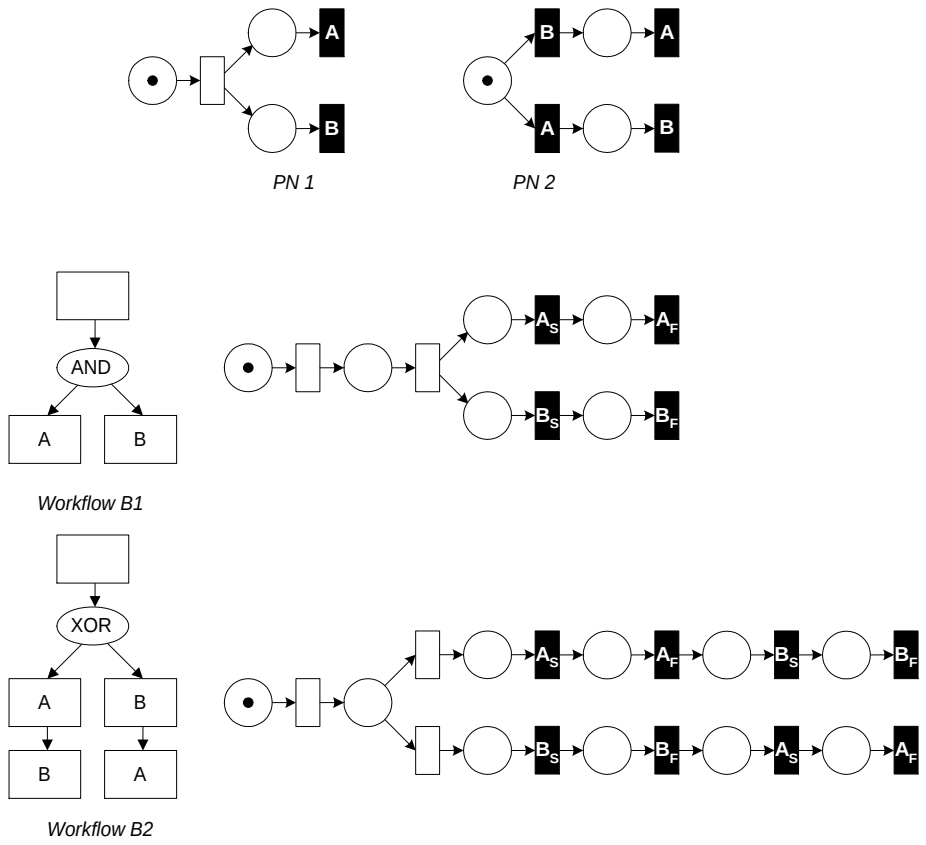

Figure 4.14: Interleaving vs. concurrent activity invocation

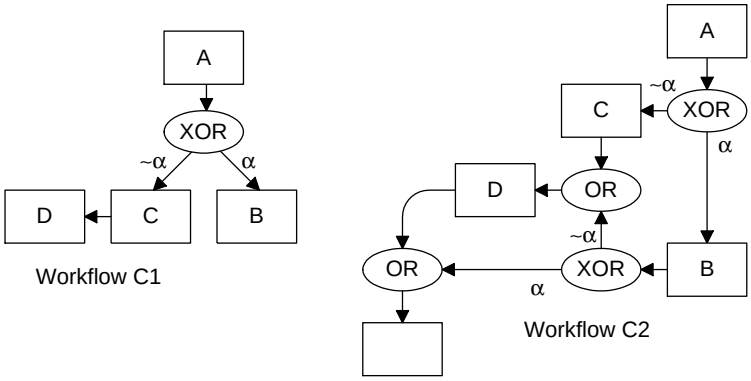

Figure 4.15: Equivalence in the context of data flow

not take conditions of XOR-Splits into account and Petri net representations of these two workflows are not equivalent (as in Workflow $C2$, after executing activity B it is still possible to fire transition D, which is not a possibility in Workflow $C1$). At this

point the careful reader may notice that in real-world workflows activity B can change the value of α and, indeed, in Workflow $C2$ it may be possible to invoke activity D after activity B. As our thesis focuses exclusively on control flow and we do not take data into consideration (except for the discussion in section 6.5), we are assuming that these two workflows are not equivalent as we have no knowledge about the possible interdependencies between the two XOR-Splits in Workflow $C2$. In other words we are always treating XOR-Splits as *non-deterministic* constructs, i.e. any decision can always be taken at any point in time.

So far we have explored different notions of equivalence in a very informal manner. Our goal was to choose an equivalence notion that is relatively simple yet powerful enough to be able to distinguish workflows that need to be considered "different". To be able to establish theoretical expressiveness boundaries of different workflow classes, we need to define our equivalence notion in a formal, precise manner.

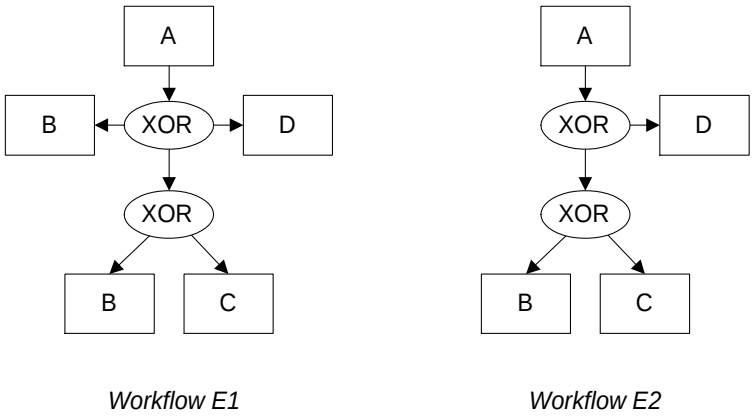

Figure 4.16: Weak bisimulation vs. branching bisimulation

The standard equivalence notion that is based on the interleaving assumption and preserves branching time is that of *bisimulation*. Bisimulation is extensively studied, primarily in the context of process graphs but also in the context of Petri nets. As the Petri nets that correspond to workflow models contain many silent transitions, focus is on *weak* bisimulation, where one abstracts from silent steps, i.e., silent steps may be executed but their execution is not visible for an external observer. As pointed out by van Glabbeek in [Gla94], Milner's notion of *weak bisimulation* in [Mil89] does not actually preserve branching time for silent transitions. This observation led to his introduction of the notion of *branching bisimulation*. Consider for example the two workflows of Figure 4.16. They are equivalent under Millner's weak bisimulation notion however they are different under van Glabbeek's branching bisimulation notion due to the fact that in workflow $E1$ there is a point where the observable run of ab diverges from the runs of ac and ad which is not the case in workflow $E2$. From a

workflow point of view we would like to consider these two workflows to be equivalent due to the fact that there is no additional data available for the second XOR-Split, therefore the moment of choice for activity B is irrelevant.

Finally, it is important that the equivalence notion distinguishes processes that successfully terminate from the ones that deadlock. As Millner's classical definition of bisimulation does not take it into account, our definition differs slightly from it, and was adapted from [Jan94].

Before we introduce bisimulation formally, we would like to present a weaker equivalence notion, namely *simulation*. Understanding simulation equivalence helps with understanding bisimulation equivalence and sometimes proving simulation equivalence precedes proving bisimulation. Processes that are bisimulation equivalent are also simulation equivalent, but the reverse does not always hold.

Definition 4.2.1

Let $PN = \langle P, T, F, L\rangle$ be a Petri net where L is a mapping that associates to each transition $t \in T$ a label $L(t)$ taken from some given set of actions $\mathcal{A}$. For any $a \in \mathcal{A}, M \stackrel{a}{\Longrightarrow}_{PN} M'$ means that $M \stackrel{\sigma}{\longrightarrow}_{PN} M'$ for some sequence σ of transitions, one of them being labelled with a, the others with λ; in case $a = \lambda$, the sequence can be empty. □

Definition 4.2.2 (*simulation*)

Given two labelled Petri nets $PN_1 = \langle P_1, T_1, F_1, L_1\rangle$ and $PN_2 = \langle P_2, T_2, F_2, L_2\rangle$, a binary relation $R \subseteq \mathbb{N}^{P_1} \times \mathbb{N}^{P_2}$ is a simulation *iff*

1. *For all $(M_1, M_2) \in R$ and for each $a \in \mathcal{A}$ and M_1' such that $M_1 \stackrel{a}{\Longrightarrow}_{PN_1} M_1'$ there is M_2' such that $M_2 \stackrel{a}{\Longrightarrow}_{PN_2} M_2'$ and $(M_1', M_2') \in R$*
2. *$(M_1, M_2) \in R \Rightarrow (M_1 \longrightarrow M_{PN_1}^{\varnothing} \Rightarrow M_2 \longrightarrow M_{PN_2}^{\varnothing})$*

Net system (PN_1, M_0) can be simulated by net system (PN_2, M_0') if there is a simulation relation R relating their initial markings.

Two labelled net systems (PN_1, M_0) and (PN_2, M_0') are simulation equivalent *if (PN_1, M_0) can be simulated by (PN_2, M_0') and (PN_2, M_0') can be simulated by (PN_1, M_0).* □

Definition 4.2.3 (*weak bisimulation*)

Given two labelled Petri nets $PN_1 = \langle P_1, T_1, F_1, L_1\rangle$ and $PN_2 = \langle P_2, T_2, F_2, L_2\rangle$, a binary relation $R \subseteq \mathbb{N}^{P_1} \times \mathbb{N}^{P_2}$ is a bisimulation *iff*

1. *For all $(M_1, M_2) \in R$:*

(a) *For each* $a \in \mathcal{A}$ *and* M_1' *such that* $M_1 \stackrel{a}{\Longrightarrow}_{PN_1} M_1'$ *there is* M_2' *such that* $M_2 \stackrel{a}{\Longrightarrow}_{PN_2} M_2'$ *and* $(M_1', M_2') \in R$, *and conversely*

(b) *For each* $a \in \mathcal{A}$ *and* M_2' *such that* $M_2 \stackrel{a}{\Longrightarrow}_{PN_2} M_2'$ *there is* M_1' *such that* $M_1 \stackrel{a}{\Longrightarrow}_{PN_1} M_1'$ *and* $(M_1', M_2') \in R$.

2. $(M_1, M_2) \in R \Rightarrow (M_1 \longrightarrow M_{PN_1}^{\varnothing} \Leftrightarrow M_2 \longrightarrow M_{PN_2}^{\varnothing})$

Two labelled net systems are bisimilar *if there is a (weak) bisimulation relating their initial markings.* □

Definition 4.2.4 (*begin-end transformation*)
Given a labelled Petri net $PN = \langle P, T, F, L \rangle$ *and* $T^l = \{t \in T \mid L(t) \neq \lambda\}$, *the net* $PN^* = \langle P', T', F', L' \rangle$ *with*

$$\begin{array}{rcl} P' & = & P \cup \{p_t \mid t \in T^l\}, \\ T' & = & T \cup \{s_t | t \in T^l\} \cup \{f_t | t \in T^l\} - T^l \\ F' & = & F[P' \times T' \cup T' \times P'] \cup \\ & & \{(p, s_t) \mid p \in {\bullet t} \wedge t \in T^l\} \cup \{(s_t, p_t) \mid t \in T^l\} \cup \\ & & \{(p_t, f_t) \mid t \in T^l\} \cup \{(f_t, q) \mid q \in t \bullet \wedge t \in T^l\} \\ L' & = & \{(t, \lambda) \mid t \notin T^l\} \cup \{(s_t, L(t)_{\mathsf{S}}) \mid t \in T^l\} \cup \{(f_t, L(t)_{\mathsf{F}}) \mid t \in T^l\} \end{array}$$

is the begin-end transformation of PN. □

Definition 4.2.5 (*workflow equivalence*)
Workflow models $\mathcal{W}_1$ *and* $\mathcal{W}_2$ *are* equivalent *iff the begin-end transformations of their corresponding net systems are bisimilar.* □

Sometimes we will compare workflow models with net systems. In that case we will say that a workflow model $\mathcal{W}$ is equivalent to a net system PN iff the begin-end transformations of the corresponding net system of $\mathcal{W}$ and PN are bisimilar.

Another, simpler and more intuitive (albeit less formal) way of looking at bisimulation is through a so-called *bisimulation game*. Below we present a bisimulation game in the context of two workflow processes, adapted from [Jan94] where it was presented in the context of Petri nets:

1. There are two players, Player A and Player B, each of which having a workflow model specification (Workflow A and Workflow B respectively).

2. Player A starts the initial activities in his workflow model specification. Player B responds by starting the initial activities in his workflow model specification (which should correspond exactly to those of player A).

3. Player A may choose to finish any of its activities and start a corresponding subsequent activity. Player B responds accordingly by finishing and starting an activity with the same label (possibly performing some internal, non-labelled, steps first).

4. If Player B cannot imitate the move of Player A, he looses. By imitating we mean that at any point in time the same set of activities in workflow B should be completed and started as in workflow A.

5. After each move, players can "switch sides", and instead of player B imitating player A, it is player's A challenge to imitate player B's move.

6. Player B wins if he can terminate his workflow once Player A has terminated his workflow. Similarly Player B wins if he can deadlock his workflow once Player A has deadlocked his workflow. The case of an infinite run of the game is considered to be successful for the defending player.

If there is a strategy for defending player (Player B) to always prevent Player A from winning then we say that workflow B can simulate workflow A. If the reverse applies as well (workflow A can simulate workflow B) then we consider the two workflow specifications to be equivalent.

4.3 Summary

In this chapter we have provided the theoretical foundation for establishing expressiveness results related to four different classes of workflow modelling techniques. It is important to understand that participation in one of these classes is not a full characterization of a given language's expressive power. There are many other factors to consider, and some of these are presented in more detail in Chapter 6. However, these classes try to capture the underlying philosophy behind given workflow engines.

Table 4.1 provides a classification of the workflow engines evaluated in this thesis according to the four classes introduced in this chapter.

Product	**Evaluation Strategy**
FileNet's Visual WorkFlo	Structured
Forté Conductor	Standard
HP Changengine	Safe
Staffware	Safe
Fujitsu i-Flow	Safe
MQSeries Workflow	Synchronised
Verve	Standard
SAP R/3 Workflow	Structured

Table 4.1: Classification of workflow products according to evaluation strategy

Chapter 5

Basic Expressiveness Results

This chapter will establish precise characterizations of the expressive power of Standard Workflow Models (Section 5.1), Safe Workflow Models (Section 5.2), Structured Workflow Models (Section 5.3) and Synchronizing Workflow Models (Section 5.4).

5.1 Standard Workflow Models

This section focuses on the expressive power of Standard Workflow Models. It is easy to verify that the corresponding Petri net system of a Standard Workflow Model is free-choice and one may wonder if these models have the same expressive power as free-choice Petri nets. This turns out not to be true. Standard Workflow Models are in fact less expressive than free-choice Petri nets. This result is not merely of theoretical importance. We will show that Pattern 16 (Deferred Choice) introduced in Chapter 3 is not possible to model using Standard Workflow Models.

Theorem 5.1.1 Standard Workflow Models are less expressive than free-choice Petri nets.

Proof:

First observe that any corresponding net of a Standard Workflow Model is free-choice. To complete the proof, we have to find a free-choice Petri net that does not have an equivalent Standard Workflow net. Such a net is shown in Figure 5.1. As can be seen, this free-choice Petri net, which we will refer to as PN_d, is very simple, yet its inherent properties may be overlooked in workflow analysis.

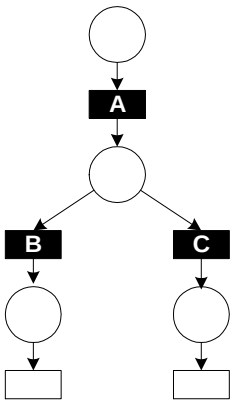

Figure 5.1: Free-choice Petri net with deferred choice

Suppose there exists a Standard Workflow net, say PN_s, equivalent to PN_d. Let us focus on marking M_1^d where there is a token in the place input to the transitions labelled B and C. If PN_d is to be bisimulation equivalent to PN_s, there should be a marking M_1^s that is related through the bisimulation relation to M_1^d (see Figure 5.2). The first observation is that in M_1^s it is not possible that both B and C are enabled. The reason for this is that although markings in Standard Workflow Models can exist which enable more than one labelled transition, it is not possible that the firing of one labelled transition leads to other labelled transitions being disabled. Hence, if both transitions B and C are enabled in M_1^s, they will both be executed at some stage, and as this is not the case for M_1^d, these two markings cannot be related through a bisimulation relation.

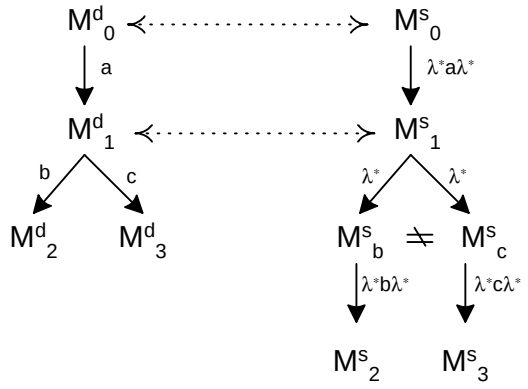

Figure 5.2: Illustration of bisimulation relations between markings

For M_1^s then to be related to M_1^d through the bisimulation relation, it should

be possible to reach markings that enable B and markings that enable C. As transitions labelled B and C cannot be enabled at the same time, we have that at least one silent step is needed (from M_1^s) to reach either a marking in which a transition labelled B is enabled or a marking in which a transition labelled C is enabled. Without loosing generality, we can assume that at least one silent step is needed to reach a marking in which a transition labelled B is enabled. Let us refer to such a marking as M_b^s. Through the bisimulation relation, this particular marking has to be related to marking M_1^d in PN_d. However, in M_1^d the transition labelled C is enabled, while in M_b^s C cannot be performed anymore. Contradiction. □

Naturally, the previous result immediately raises the question as to what the exact expressive power of Standard Workflow Models is. Before we provide a complete characteristic of the expressive power of Standard Workflow Models let us focus on some of the most basic properties of theses models.

The following lemma states that once a process element becomes enabled, it cannot be disabled by firing any other process element but itself and can be proved by case distinction.

Lemma 5.1.1 Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ be a Standard Workflow Model and $e, p \in \mathcal{P}$ enabled process elements of $\mathcal{W}$ in a given instance of $\mathcal{W}$ ($e \neq p$). After firing p, e is still enabled.

From all the process elements only activities contain labelled transitions. The next theorem proves that for a free-choice Petri net to have a bisimulation equivalent Standard Workflow Net it is sufficient that all its labelled transitions, once they become enabled, cannot be disabled by firing any other transitions but themselves.

We will refer to such a subclass of free-choice Petri nets as Free-Choice Deterministic Action Nets.

Definition 5.1.1

A Free-Choice Deterministic Action Net *(FCDA net)* $PN = \langle P, T, F, L \rangle$ *is a labelled free-choice Petri net, i.e.*

$$\forall_{t \in T, p \in P} \left[(p,t) \in F \Rightarrow {\bullet t} \times p{\bullet} \subseteq F \right],$$

where every labelled transition has exactly one input place and that place is not an input to any other transition:

$$\forall_{t \in T} [L(t) \neq \lambda \Rightarrow \forall_{t' \in T} \left[{\bullet t} \cap {\bullet t'} \neq \varnothing \Rightarrow t = t' \right]].$$

□

Theorem 5.1.2 Standard Workflow nets are as expressive as FCDA net systems.

Proof:

As every Standard Workflow net is an FCDA net, we can focus on proving that every FCDA net has a bisimilar Standard Workflow net. This will be achieved in a constructive way, i.e. the proof will focus on the translation of any arbitrary FCDA net to a Standard Workflow net. The organization of the proof is as follows: given an FCDA net, PN we will perform a number of bisimulation-preserving transformations on it eventually deriving a net, PN_1. At the same time we will construct a Standard Workflow Model $\mathcal{W}$ for which its corresponding Petri net $PN_{\mathcal{W}}$ is identical to PN_1. This will conclude the proof.

The translation takes a number of steps. In intermediate stages, instead of a pure Petri net notation we will use a shorthand representation of Petri net subnets using workflow construct notation. This serves two purposes: (1) it dramatically simplifies the complexity of the derived net and (2) it allows us to construct the desired Standard Workflow Model. An example of a shorthand notation is shown in Figure 5.3, which shows three places linked to a hybrid Activity and AND-Join construct. AND-Split, XOR-Split and OR-Join constructs are derived in a similar manner. All presented translations will make sure that hybrid constructs will always be linked to places or to each other. Let us define two sets, T^* and P^* representing the sets of transitions and places respectively that are part of the hybrid net but not part of a hybrid structure. Initially $T^* := T$ and $P^* := P$. Each transformation step aims to reduce the number of elements in T^* or P^* (or both) until all transitions and places are part of a hybrid structure. For example, in Figure 5.3 $T^* = \varnothing$ while $P^* = \{P1, P2, P3\}$.

For the construction to be meaningful, it is required that every transformation step preserves equivalence. This is easy to check for each of the steps presented.

The following steps describe the procedure to transform any arbitrary FCDA system into a bisimulation equivalent Standard Workflow net.

1. Replace all places with initial tokens with the structure shown in diagram (a) of Figure 5.4. The number of null activities should correspond to the number of tokens. If there is only one token then the OR-Join is redundant and can be omitted. After this step there are no tokens in any of the places of the net. This step does not affect T^* or P^*.
2. A labelled transition has exactly one input place that is not shared with any other transition. Diagram (a) of Figure 5.5 presents the transformation for a labelled transition with one output place and diagram (b) of that Figure presents the transformation for a labelled transition with many output

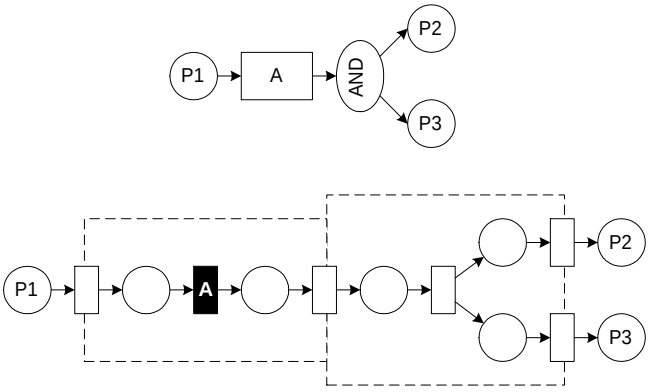

Figure 5.3: Interpretation of a sample hybrid net

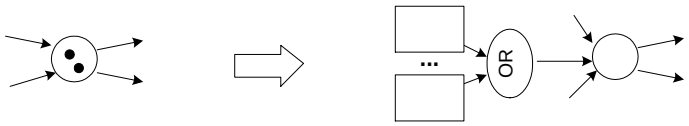

Figure 5.4: Translation of marked places

places. After this step, there are no labelled transitions anymore, i.e. $T^* := \{t \in T^* \mid L(t) = \lambda\}$.

3. Replace transitions with no input or output places and places with no input or output transitions by corresponding structures as shown in Figure 5.6. Note that the semantics of Splits without incoming transitions and Joins without outgoing transitions is such that these transformations are equivalence preserving. After that step

$$T^* = \{t \in T \mid L(t) = \lambda \wedge |t\bullet| \geq 1 \wedge |\bullet t| \geq 1\}$$
$$P^* = \{p \in P \mid |p\bullet| \geq 1 \wedge |\bullet p| \geq 1\}$$

4. Replace transitions that have the same, nonsingular, set of input places

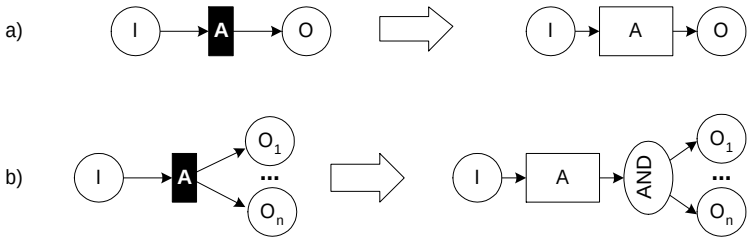

Figure 5.5: Translations of labelled transitions

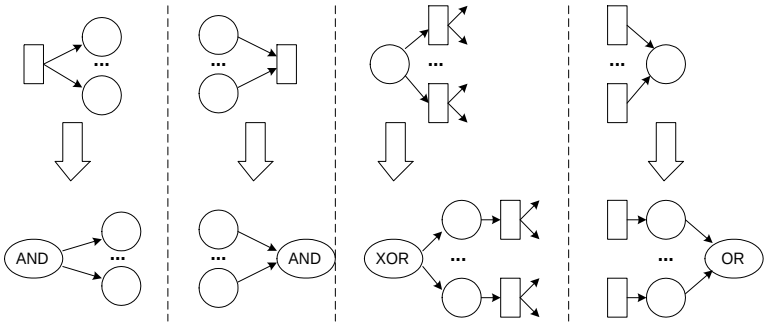

Figure 5.6: Translations of transitions/places without input or output

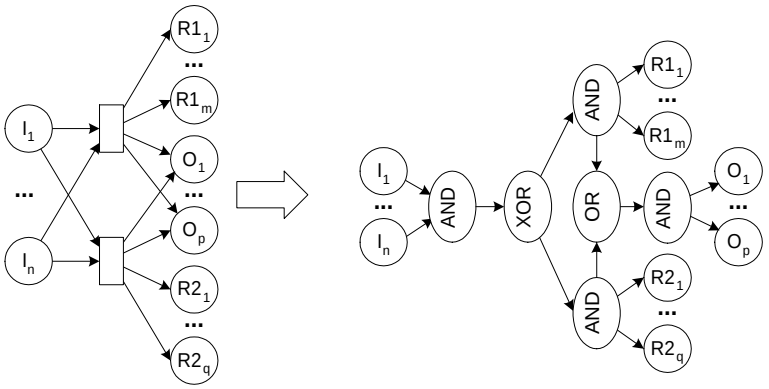

Figure 5.7: Removal of transitions sharing nonsingular set of input places

with the structure shown in Figure 5.7. Effectively, from this step onwards, if transitions share any input places, they share exactly one (remember that an FCDA net is free-choice). Note that if any of the transitions have only one output place, the AND-Split can be omitted. Formally we now have that

$$\begin{aligned} T^* &= \{t \in T \mid L(t) = \lambda \wedge |t\bullet| \geq 1 \wedge |\bullet t| \geq 1 \wedge \\ &\qquad \forall_{t' \in T} [\bullet t \cap \bullet t' \neq \varnothing \Rightarrow (t = t' \vee |\bullet t| = 1)]\} \\ P^* &= \{p \in P \mid |p\bullet| \geq 1 \wedge |\bullet p| \geq 1\} \end{aligned}$$

5. At this stage it is still possible that transitions share input places, output places, or both. In Figure 5.8 the removal of such transitions is defined in diagrams (a), (b) and (c). Again, in all these transformations, if any of the transitions have only one input or one output place, the AND-Joins and

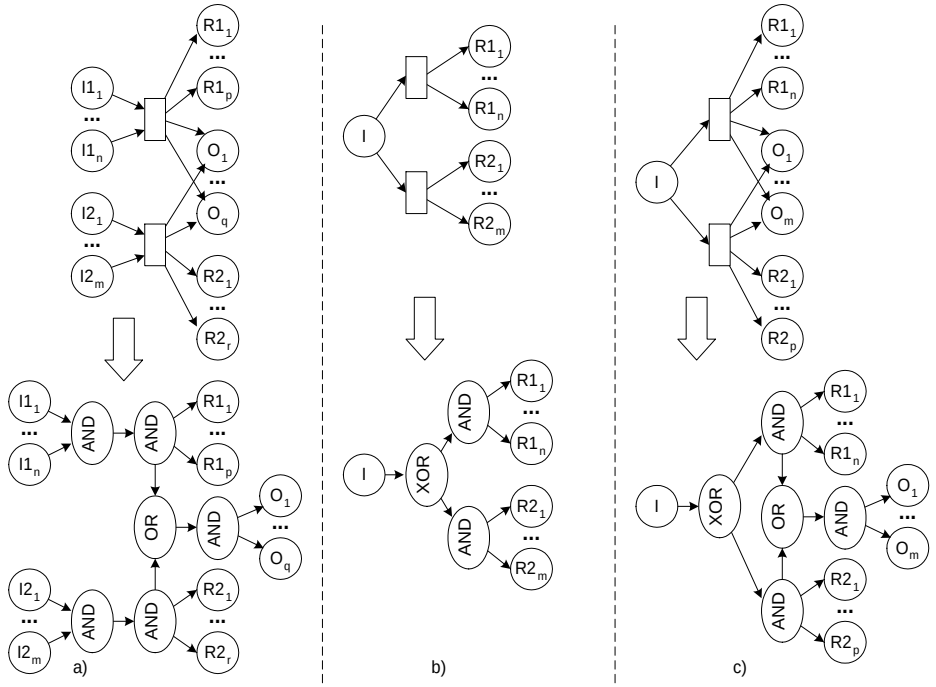

Figure 5.8: Removal of transitions sharing input or output places

AND-Splits respectively can be omitted. After this step:

$$\begin{aligned} T^* &= \{t \in T \mid L(t) = \lambda \wedge |t\bullet| \geq 1 \wedge |\bullet t| \geq 1 \wedge \\ &\qquad \forall_{t' \in T} \left[(\bullet t \cap \bullet t' \neq \varnothing \vee t\bullet \cap t'\bullet \neq \varnothing) \Rightarrow t = t'\right]\} \\ P^* &= \{p \in P \mid |p\bullet| \geq 1 \wedge |\bullet p| \geq 1\} \end{aligned}$$

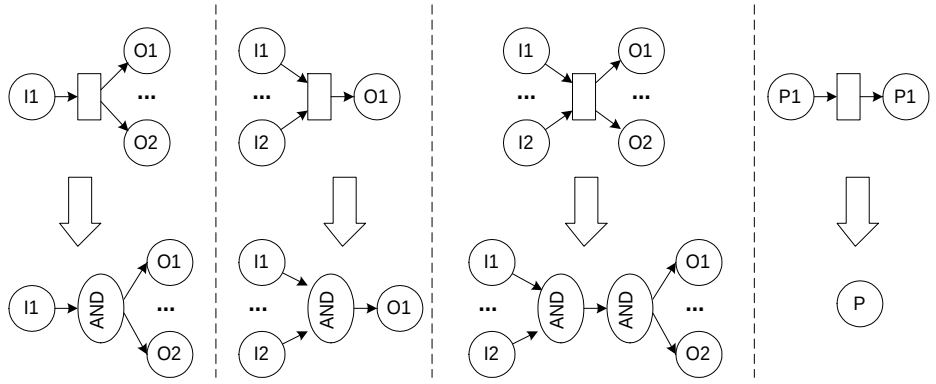

Figure 5.9: Removal of transitions

6. In this step all remaining transitions are removed as shown in Figure 5.9. There are four possibilities - the transition may have one input place and many output places, many input places and one output place, many input and many output places, or one input place and one output place. After this step
$$\begin{aligned} T^* &= \varnothing \\ P^* &= \{p \in P \mid |p\bullet| \geq 1 \wedge |\bullet p| \geq 1\} \end{aligned}$$

7. Now that all transitions are removed, places can only be linked to workflow constructs. They can subsequently be removed according to the schema shown in Figure 5.10. Again, as with the previous step, there are only four possibilities. After this step the net consists entirely of hybrid constructs, i.e. $T^* = \varnothing$ and $P^* = \varnothing$.

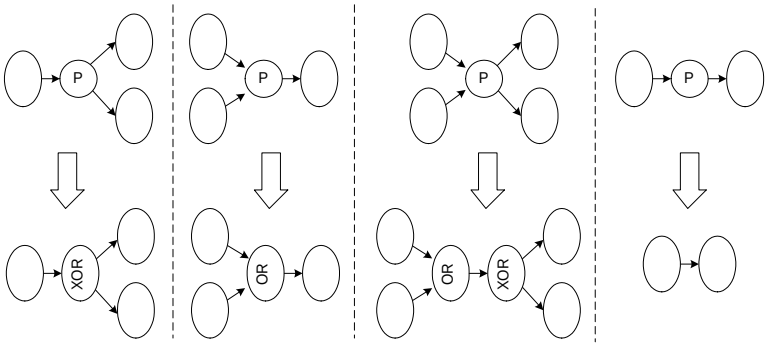

Figure 5.10: Removal of places

As every step that we have taken so far is equivalence preserving, the hybrid net that we have constructed, PN_H, is equivalent to our source FCDA net, PN. As PN_H consists entirely of hybrid structures, it is possible to construct a Standard Workflow Model $\mathcal{W}$ by replacing hybrid structures with the corresponding workflow constructs. The corresponding Petri net PN_W of the Standard Workflow Model $\mathcal{W}$ constructed in such a manner is identical to PN_H. As PN_H is equivalent to PN, it follows that W is equivalent to PN which concludes the proof.

□

Example 5.1.1 An example of the transformation described in the proof of Theorem 5.1.2 of an FCDA net to a Standard Workflow Model is shown in Figure 5.11. Obviously, the Standard Workflow Model can be further reduced (the final AND-Join is redundant and can be removed), however, this is of no importance in this context. Note that the FCDA net and the Petri net corresponding

to the Standard Workflow Model (see Figure 4.5) are indeed weak bisimulation equivalent. □

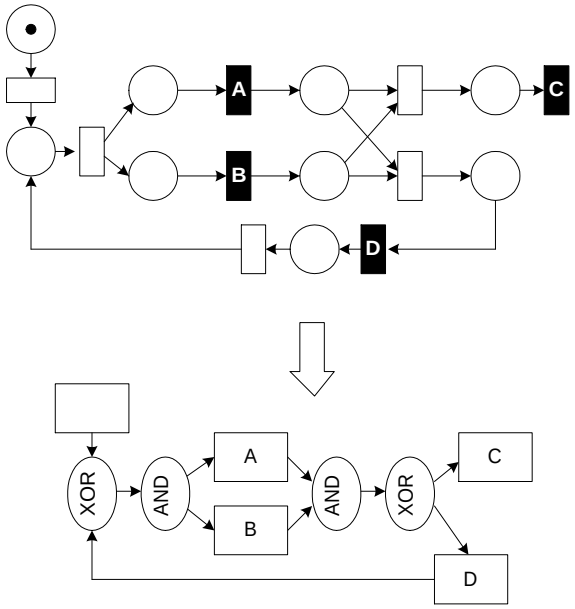

Figure 5.11: FCDA net with equivalent Standard Workflow Model

5.2 Safe Workflow Models

As explained in Section 4.1, the main difference between Standard Workflow Models and Safe Workflow Models is in the interpretation of the OR-Join in case it is triggered by more than one incoming branch (as could e.g. happen in case an OR-Join follows an AND-Split).

In this section we would like to answer the question whether this evaluation strategy limits the expressive power of the workflow engine. Formally, this translates to the question as to whether it is possible to transform any given Standard Workflow Model to an equivalent Safe Workflow Model. A technique typically required for this is node replication (illustrated in Figure 5.12).

Node replication can be compared to net unfolding as described in for example [GV87]. The unfolded net can be thought of as the safe version of the original net. Unfolding as described in [GV87] preserves bisimulation equivalence.

It is immediately clear that if the original net is not bounded, then the unfolding is infinite. Hence, it is impossible to convert a Standard Workflow Model that may

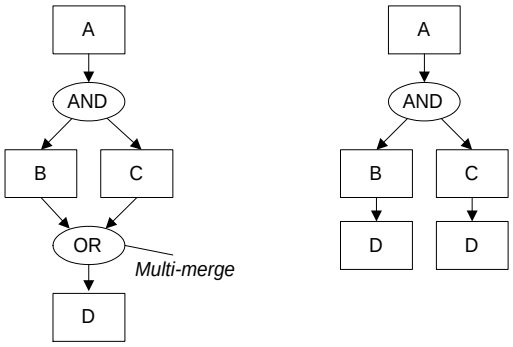

Figure 5.12: Node replication

result in an unlimited number of multiple instances of some activity, into a finite Safe Workflow Model. Therefore, let us focus on bounded workflow models.

It is always possible to convert a bounded Petri net into an equivalent safe Petri net by unfolding. However, from a workflow perspective, the fundamental problem with this technique is that unfolding as presented in [GV87] may transform a free-choice Petri net into a net which is not free-choice. The next theorem demonstrates that this is a true problem which cannot be circumvented. There exist bounded Standard Workflow specifications that do not have a safe equivalent.

Before presenting a proof we would like to introduce two lemmas. The first one captures one of the important characteristics of free-choice nets. This lemma will be used in several subsequent proofs and it states that if there is a path from a place q to a place p, and $[p]$ is a home marking (i.e. a marking which is reachable from every reachable marking), then if q contains a token it can be moved to p by a firing sequence containing all transitions on the path between q and p.

Lemma 5.2.1 Let $PN = (P, T, F, M_0)$ be a live and bounded free-choice Petri net with a home marking $M_0 = [p]$ (i.e. the state marking a place p). Let M be a reachable marking which marks place q and let $x =< p_1, t_1, p_2, t_2, ..., t_{n-1}, p_n >$ with $p_1 = q$ and $p_n = p$ be an acyclic directed path. Then there is a firing sequence σ such that $M \xrightarrow{\sigma} [p]$, each of the transitions $\{t_1, ..., t_{n-1}\}$ is executed in the given order, and none of the intermediate markings marks p.

Proof:

If $p = q$ then the lemma holds. If $p \neq q$ then there is a firing sequence removing the token from q (since $[p]$ is a home marking). Let $\sigma_1 t$ be the firing sequence removing the token from q, i.e. $t \in q\bullet$. Let M_1 be the marking enabling t, i.e. $M \xrightarrow{\sigma_1} M_1$. As the net is free-choice and t is enabled in M_1, t_1 is also enabled

in M_1 (recall that $q \in \bullet t_1$ and $q \in \bullet t$ implies $\bullet t_1 = \bullet t$). It is therefore possible to fire t_1, i.e. $M_1 \xrightarrow{t_1} M_2$. In M_2 place p_2 is marked (as $p_2 \in t_1\bullet$).

By recursively applying the argument to the remaining places and transitions it is possible to construct a firing sequence σ such that each transition in $\{t_1, ..., t_{n-1}\}$ occurs and $M \xrightarrow{\sigma} [p]$, i.e. it is possible to execute the transitions in the order of the directed path between q and p.

Remains to prove that none of the intermediate markings reached by executing σ marks p. Suppose that p was marked before completing σ. There is a token moving from q to p via path $< q, t_1, p_2, t_2, ..., t_{n-1}, p >$. Therefore, for any intermediate marking there is a token in one of the places $\{p_2, ..., p_{n-1}\}$. However, if p and some other place are marked at the same time the net is unbounded. This contradiction completes the proof. □

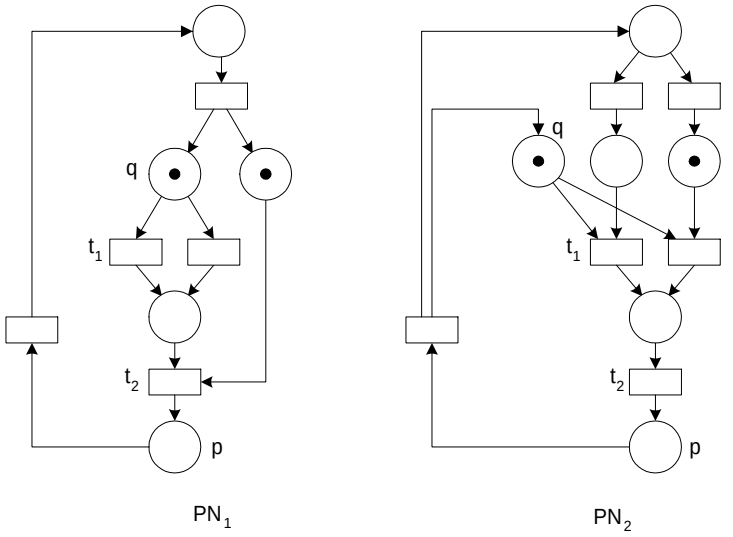

Figure 5.13: Illustration of Lemma 5.2.1

Figure 5.13 illustrates two Petri nets, PN_1 being a free-choice net, and PN_2 not. Place p is a home marking for both nets. Let us concentrate on the path from place q to place p containing transitions t_1 and t_2. In net PN_1, from any marking having a token in place q it is possible to fire transitions t_1 and t_2. In net PN_2 that is not always the case as the marking shown illustrates.

The second lemma introduces a construct that we would like to refer to as a "selective synchronizer". Such a synchronizer has three incoming transitions. In the context shown in Figure 5.14 the Selective Synchronizer awaits completion of activity A and either activity B or activity C. Depending on whether B or C completes, activity D

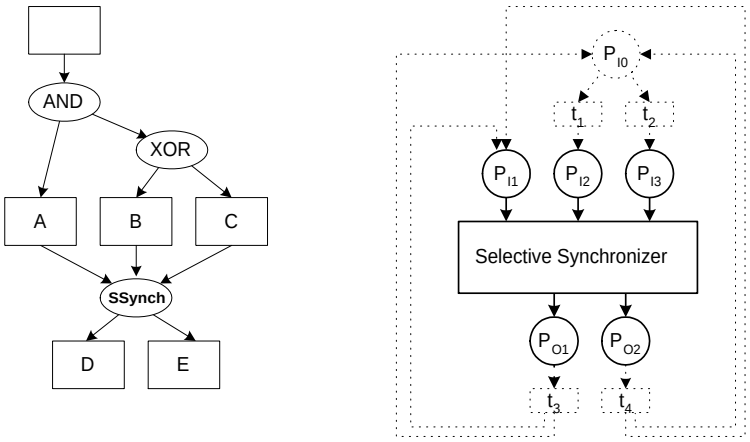

Figure 5.14: Illustration of Lemma 5.2.2

or activity E respectively is enabled. It is worth noticing that the desired behaviour is not achievable using standard workflow constructs. For example, had we put an OR-Join after activities B and C, it would not be possible to make a correct choice between D and E. On the other hand any attempts to use a standard AND-Join construct leads to a deadlock.

The following lemma defines the "selective synchronizer" in a formal way and proves that this construct is inherently non free-choice.

Lemma 5.2.2 Let $PN = \langle P, T, F \rangle$ be the Petri net as shown (in bold lines) in the right diagram of Figure 5.14. The Selective Synchronizer construct cannot be free-choice if:

- Any marking with tokens in any of the places p_{O1}, or p_{O2} has one token in exactly one of these places and no other places. Such markings are called *output markings*;
- – From $p_{I1} + p_{I2}$, the only reachable output marking is p_{O1};
  – From $p_{I1} + p_{I3}$, the only reachable output marking is p_{O2}.

Proof:

Consider the Selective Synchronizer net augmented with place p_{I0}, transitions t_1, t_2, t_3 and t_4 and arrows as shown with dashed lines in Figure 5.14. The resulting Petri net is called the short-circuited net. Clearly, $[p_{I1}, p_{I2}]$, $[p_{I1}, p_{I3}]$, $[p_{O1}]$ and $[p_{O2}]$ are home markings. We can assume that the Selective Synchronizer construct contains no dead transitions and that the short-circuited net is strongly connected. Places and transitions without any input and/or output

arcs are either inactive and do not contribute to the external behaviour or are conflicting with the requirements. As a result, the short-circuited net is live and bounded with home markings $[p_{I1}, p_{I2}]$, $[p_{I1}, p_{I3}]$, $[p_{O1}]$ and $[p_{O2}]$.

As the marking $p_{I1} + p_{I2}$ is followed by $[p_{O1}]$, we can conclude that there must be a path from p_{I1} to p_{O1} and from p_{I2} to p_{O1}. Similarly there must be a path from p_{I1} to p_{O2} as the marking $p_{I1} + p_{I3}$ is followed by $[p_{O2}]$.

Suppose that the selective synchronizer is a free-choice construct. The whole net is then free-choice too. According to Lemma 5.2.1 if there is a path from p_{I1} to p_{O2}, then there is also a firing sequence leading from $p_{I1} + p_{I2}$ to p_{O2}. But this is contradictory with the assumptions. □

Finally we are ready to present a theorem that shows the expressiveness limitation of the safe evaluation strategy.

Theorem 5.2.1 (*limited power of the safe evaluation strategy*) There exist bounded Standard Workflow Models without a deadlock, for which there exists no equivalent Safe Workflow Model.

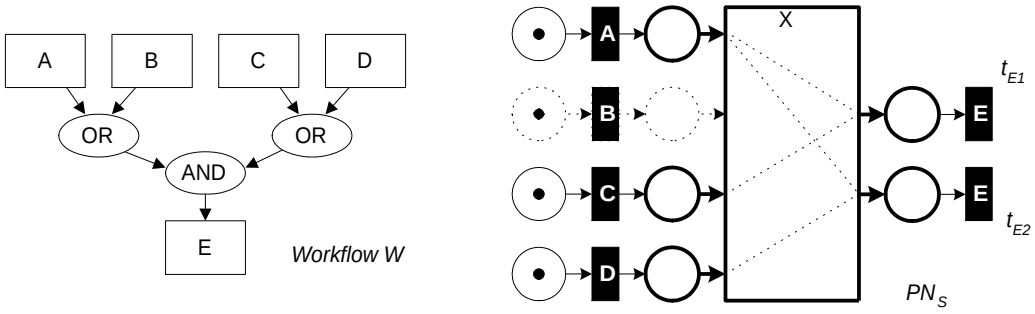

Figure 5.15: Multiple instances specification

Proof:

Consider the deadlock free and bounded Standard Workflow Model $\mathcal{W}$ in Figure 5.15. There are four initial activities named A, B, C, and D. The activity named E can be fired after either A and C have been completed, or A and D, or B and C or B and D. Subsequently activity E can be fired for the second time when the remaining activities are completed.

Let $\mathcal{S}$ be a Standard Workflow Model that is bisimulation equivalent to $\mathcal{W}$ and $PN_{\mathcal{S}}$ be the corresponding net of $\mathcal{S}$. For $\mathcal{S}$ to be bisimulation equivalent to $\mathcal{W}$, $PN_{\mathcal{S}}$ needs to have transitions labelled A, B, C, and D as well as at least

two transitions labelled E. The last requirement comes from the fact that in workflow $\mathcal{W}$ it is possible to enable and fire the transition labelled E twice in a concurrent manner.

In $\mathcal{W}$ it is possible to enable activities A, B, C and D concurrently. Hence there must be a reachable marking M of $PN_{\mathcal{S}}$ that enables transitions labelled A, B, C and D and no other labelled transitions. Let us call these transitions t_A, t_B, t_C and t_D respectively.

In $\mathcal{W}$ it is possible to fire activities A and C followed by activity E. Thus in net PN_S there must be a path from t_A and t_C to a transition labelled E. Let us call this transition t_{E1}.

Similarly there must be paths from transitions t_A and t_D to a transition labelled E as well as paths from t_B, t_C and t_B, t_D to transitions labelled E. Let us call these transitions t_{E2}, t_{E3} and t_{E4} respectively.

Consider transitions t_{E1}, t_{E2} and t_{E4}. Transitions t_{E1} and t_{E4} cannot be the same (otherwise the net would not be safe) whereas it is possible that $t_{E1} = t_{E2}$ or $t_{E4} = t_{E2}$. Without loss of generality, suppose that t_{E2} is such that $t_{E1} \neq t_{E2}$.

Any labelled transition in $PN_{\mathcal{S}}$ needs to have exactly one input and one output place. Output places of transitions t_A, t_C, t_D and input places of transitions t_{E1} and t_{E2} along with the subnet X shown in the right diagram of Figure 5.15 form a subnet that fulfils the requirements of Lemma 5.2.2 (Selective Synchroniser), hence the subnet X and subsequently net PN_S cannot be free-choice. This contradicts the assumption that PN_S is the corresponding net of a Standard Workflow Model. □

Theorem 5.2.1 shows that the choice for a safe execution strategy limits the expressive power of the corresponding workflow engine, even if one is only interested in bounded deadlock-free workflows. A practical example of a process that might need the type of synchronisation shown in Figure 5.15 is a process in which activities A and B represent the manufacturing of an item of type X, activities C and D the manufacturing of an item of type Y and activity E represents the assembling of an item of type X and an item of type Y.

5.3 Structured Workflow Models

The definition of Structured Workflow Models guarantees these types of workflows to have certain properties, namely they never deadlock and they never result in multiple instances.

Theorem 5.3.1 Structured Workflow Models are deadlock free and safe.

Proof:

The proof uses structural induction on the construction of Structured Workflow Models as per Definition 4.1.15. The basis of the induction are models consisting of single activities, which obviously cannot deadlock and are safe. An investigation of Definition 4.1.15 yields that:

1. A sequential execution of two workflows that do not deadlock and are safe also does not deadlock and is safe (induction).
2. An XOR-Split and OR-Join combination does not deadlock and is safe as long as each of its branches does not deadlock and is safe (induction). Similarly for an AND-Split and AND-Join combination.
3. A loop cannot deadlock and is safe as the body of the loop cannot deadlock and is safe (again by induction).

□

As each Structured Workflow is well-behaved, the Structured Workflows form a proper subclass of Standard Workflows. An important question that needs to be answered is whether any well-behaved Standard Workflow has an equivalent Structured form. The next theorem proves that that is not the case, and Structured Workflows form a proper subset of well-behaved Standard Workflows.

Theorem 5.3.2 There are well-behaved, Standard Workflow Models that do not have an equivalent structured form.

Proof:

Consider the well-behaved Standard Workflow Model $\mathcal{W}$ shown in Figure 5.16. This workflow model contains multiple exits from a loop and as such is unstructured.

Suppose that there is a Structured Workflow Model $\mathcal{S}$ with a finite number of process elements that is bisimulation equivalent to $\mathcal{W}$. As a trace of $\mathcal{S}$ may contain an infinite number of occurrences of B and C, $\mathcal{S}$ must contain at least one structured loop. Let us refer to this loop as l.

Suppose that l does not contain any labelled activities with a label other than B. It is then possible, for each number n to produce a trace with a substring containing n consecutive occurrences of B. Such a trace is impossible in $\mathcal{W}$.

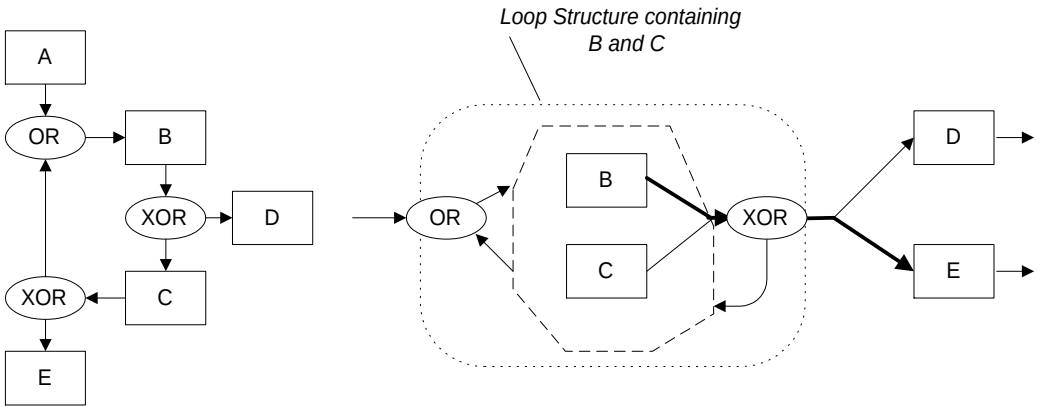

Figure 5.16: Exit from a loop structure

Similarly C cannot be the only label used for activities in l. Furthermore as all traces of $\mathcal{W}$ have at most one occurrence of A, D or E it follows that l cannot contain any activities having one of these labels. We therefore conclude that l must contain at least one activity labelled with B and at least one activity labelled with C and no activities with other labels. Let t_B be a transition labelled with B and t_C a transition labelled with C.

Consider a reachable marking M'_B of the corresponding net system of $\mathcal{S}$ such that t_B is enabled (such a marking is possible as it follows from the inductive structure of Structured Workflows that for every process element of a Structured Workflow Model there is a reachable marking that enables this process element). For $\mathcal{S}$ to be bisimulation equivalent to $\mathcal{W}$ there must be a reachable marking M'_D such that $M'_B \stackrel{b}{\Longrightarrow} M'_D$ and M'_D is a marking of $\mathcal{S}$ that enables a transition labelled D and no other labelled transitions. Let us call this transition t_D. We conclude that there must be a path from t_B to t_D that does not contain any labelled transitions.

Using the same reasoning applied to the labels C and E we can conclude that there must be a path from a transition labelled C in l to a transition labelled E that does not contain any labelled transitions. Let us call this transition t_E.

Because t_B and t_C are part of l, and t_D and t_E are not, given the structure of a Structured Workflow Model we have that in $\mathcal{S}$ there is a path from t_B to t_E that does not contain any labelled activities (this path is marked bold in the right diagram of Figure 5.16) and thus it is possible to enable t_E after completing t_B without firing any other labelled transitions. As this execution scenario is not possible in $\mathcal{W}$, this contradicts the assumption that $\mathcal{W}$ and $\mathcal{S}$ are bisimulation equivalent.

□

Having established that Structured Workflow Models are a proper subclass of well-behaved Standard Workflow Models, it is important to establish a class of Standard Workflow Models that can be transformed to a structured form. The remainder of this section focuses on to what extent such transformations are possible.

The organisation is as follows. First we concentrate on workflows that do not contain parallelism (to be more precise, we consider workflows that do not contain AND-Join and AND-Split constructs). Then we concentrate on workflows that do contain parallelism, but do not have any cycles. Finally we will consider workflow models with both loops and parallelism.

5.3.1 Simple Workflows without Parallelism

Workflows that do not contain parallelism are simple models indeed. Their semantics is very similar to elementary flow charts that are commonly used for procedural program specification. The XOR-Split corresponds to selection (if-then-else statement) while the activity corresponds to an instruction in the flow chart. Transformations of an unstructured flow chart to a structured form has been studied extensively in the past and in this section we will revisit some of these transformation techniques and present and analyse them in the context of workflow models.

Following [Wil77] we will say that the process of *reducing* a workflow model consists of replacing each occurrence of a base model (i.e. one of the four shown in Figure 4.6) within the workflow model by a single activity box. This is repeated until no further replacement is possible. A process that can be reduced to a single activity box represents a Structured Workflow Model. Each transformation of an irreducible workflow model should allow us to reduce the model further and in effect reduce the number of activities in the model.

The strong similarity of simple workflow models and flow diagrams suggests that if we do not consider parallelism, there are only four basic causes of unstructuredness (see e.g. [Wil77, Oul82]):

- Exit from a decision structure
- Entry into a decision structure
- Entry into a loop structure
- Exit from a loop structure

Entry into any structure is modelled in a workflow environment by an OR-Join construct. Similarly, an exit is modelled by a XOR-Split. Once parallelism is introduced we will also consider synchronised entry and parallel exit modelled by AND-Join and AND-Split constructs respectively.

As was shown in Theorem 5.3.2, models containing an exit from a loop structure cannot be transformed to equivalent Structured Workflow Models. For the remaining causes of unstructuredness the transformations are possible and are shown in Figures 5.17, 5.18 and 5.19 All transformations are using a technique called *node*

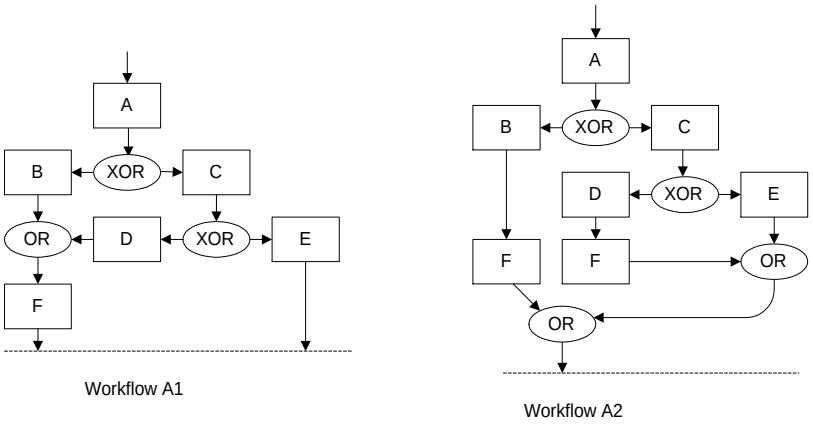

Figure 5.17: Exit from a decision structure

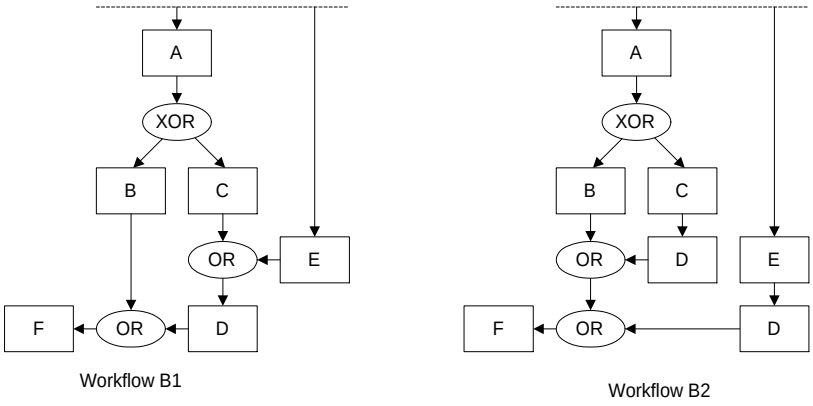

Figure 5.18: Entry into a decision structure

duplication and are adapted from [Oul82]. The transformation shown in Figure 5.17 should be used when a workflow model contains an exit from a decision structure, the

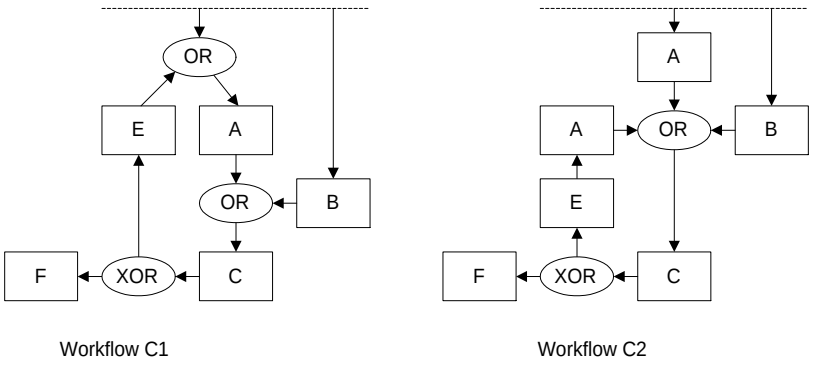

Figure 5.19: Entry into a loop structure

transformation in Figure 5.19 when a model contains an entry into a decision structure and finally the transformation in Figure 5.19 when a model contains an entry into a loop structure. It should be noted that all models shown in these figures are intended to be fragments of workflows, rather than complete workflows in themselves. The reader should verify for themselves that all transformations yield workflows that are indeed bisimulation equivalent.

As shown in [Oul82] repeated application of the transformations discussed in this section can remove all forms of unstructuredness from a workflow as long as an exit from a loop structure is not encountered. Thus all unstructured workflows without parallelism and exit from a loop structure have an equivalent structured form. Finally, it should be kept in mind that the transformations shown here use only basic control flow constructs - it is possible to derive some additional transformations using auxiliary variables, i.e. augmenting XOR-Splits with data predicates. We will come back to this issue in Section 6.5.

5.3.2 Workflows with Parallelism but without Loops

Addition of parallelism immediately introduces problems related to deadlock and multiple instances. As Theorem 5.3.1 shows, Structured Workflow Models never result in deadlock nor multiple instances of the same activity at the same time and in the remainder of this section we will focus on well-behaved Standard Workflow Models.

Theorem 5.3.2 shows that workflows containing an exit from a loop structure cannot be transformed to a structured form. As the next theorem shows, parallelism introduces other possible causes of unstructuredness that make the workflows impossible to model in a structured form.

Theorem 5.3.3 There are well-behaved Standard Workflow Models that do not contain cycles, that cannot be modelled as Structured Workflow Models.

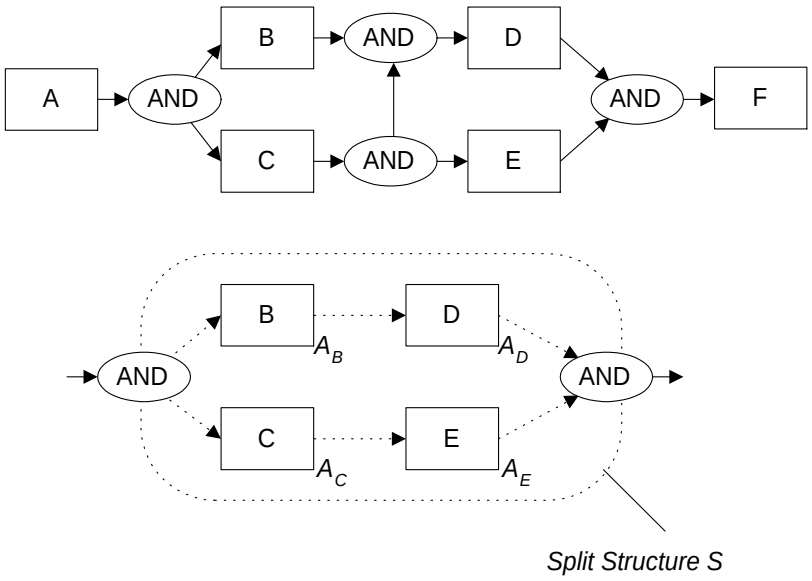

Figure 5.20: Arbitrary workflow and illustration of its essential causal dependencies

Proof:

Suppose that there is a fully structured workflow $\mathcal{S}$ that is bisimulation equivalent to $\mathcal{W}$. As it is possible in $\mathcal{W}$ that both activities B and C are simultaneously enabled, $\mathcal{S}$ must contain a split-structure such that one branch contains an activity labelled B and the other branch contains an activity labelled C (as this is the only way B and C can be simultaneously enabled in any structured workflow). Let us refer to this split-structure as s and to these activities as A_B and A_C respectively.

Applying the bisimulation game yields that there must be a path from A_C to an activity labelled E. Let us refer to this activity as A_E. As there is a marking of $\mathcal{W}$ that enables both activities B and E, there must be a marking in $\mathcal{S}$ such that A_B and A_E are both enabled. But as there is a path from A_C to A_E we conclude that A_E is on the same branch of s as A_C.

Similarly we have that there must be a path from A_B to an activity labelled D. Let us refer to this activity as A_D. As there is a marking of $\mathcal{W}$ that enables both activities D and E, there must be a marking in $\mathcal{S}$ such that A_D and A_E are both enabled. But as there is a path from A_B to A_D we conclude that A_D is on the same branch of s as A_B.

As in $\mathcal{S}$ activities D and E are in different branches of s (see Figure 5.20), there is a marking of $\mathcal{S}$ such that it enables both A_D and A_E. As there is no marking of $\mathcal{W}$ enabling both D and E, this contradicts the assumption that $\mathcal{W}$ and $\mathcal{S}$ are bisimulation equivalent.

□

To find out which Standard Workflow Models can be effectively transformed into Structured Workflow Models, let us concentrate on the causes of unstructuredness that can occur when parallelism is added. If loops are not taken into account, these causes are:

- Entry into a decision structure
- Exit from a decision structure
- Entry into a parallel structure
- Exit from a parallel structure
- Synchronised entry into a decision structure
- Parallel exit from a decision structure
- Synchronised entry into a parallel structure
- Parallel exit from a parallel structure

In the remainder of this section we will concentrate on which of these structures can be transformed to a Structured Workflow Model.

Entries and exits from decision structures are dealt with in section 5.3.1 and can obviously be transformed to a structured model.

As a synchronised entry into a decision structure and an exit from a parallel structure leads to a potential deadlock (i.e. there are instances of the model that will deadlock), it follows that if the original workflow contains any of these patterns, generally speaking it cannot be transformed into a Structured Workflow Model.

Parallel exits from and synchronised entries into a parallel structure are dealt with in theorem 5.3.3. The reasoning of this theorem can typically be applied to a model that contains these patterns. Hence such models, even though they may be well-behaved, cannot be transformed into a structured form.

Before analysing the two remaining structures let us define a syntactical structure called an *overlapping structure*. This structure has been previously introduced (and

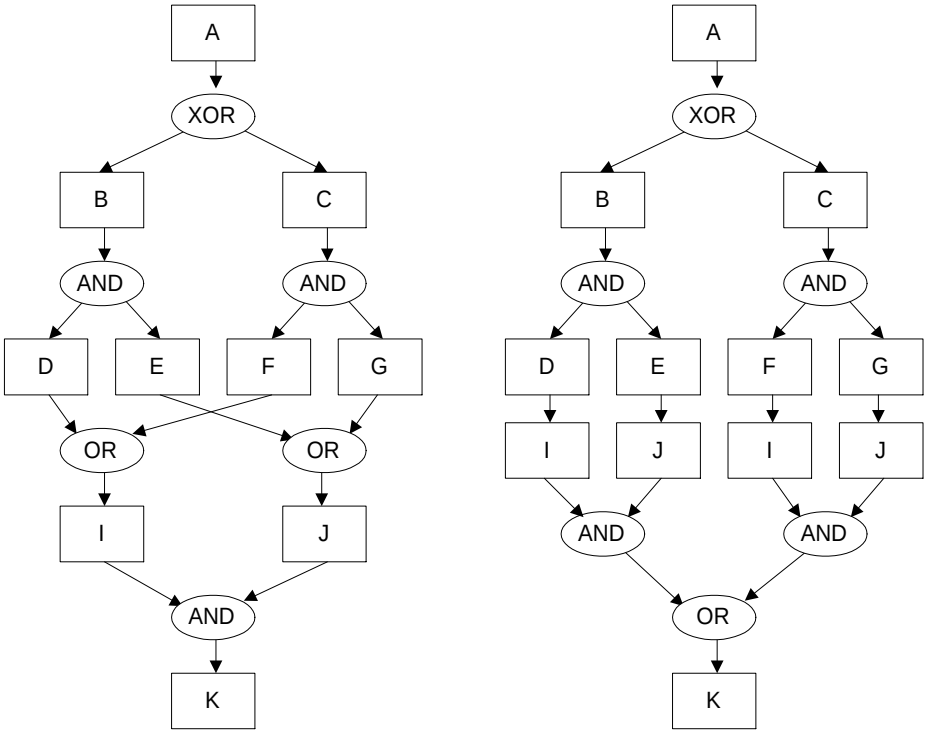

Figure 5.21: Overlapping structure

named) in the context of workflow reduction for verification purposes in [SO99]. A specific instance of it is shown in Figure 5.21. An overlapping structure consists of a XOR-Split followed by i instances of AND-Splits, followed by j instances of OR-Joins and finally by an AND-Join. The structure of Figure 5.21 has both i and j degrees equal to two. The overlapping structure contains both an entry into a parallel structure and a parallel exit from a decision structure and it never results in a deadlock. It is possible to transform an overlapping structure into a Structured Workflow Model as shown in Figure 5.21.

In [SO99] an analysis of the causes of deadlock and multiple instances for a subclass of Standard Workflow Models is provided. This subclass consists of workflows that do not contain cycles and have only one final task. This work leads to the conclusion that Standard Workflow Models that have only one final task and contain a parallel exit from a decision or an entry into a parallel structure will cause a potential deadlock or multiple instances unless these causes of unstructuredness are part of an overlapping structure. In [LZLC02] and [AHV02] it has been shown that more complex structures similar to an overlapping structure exist. The examples provided in both [LZLC02] and [AHV02] can also be easily tranformed to an equivalent structured form. These observations have led us to the following conjecture:

Conjecture 5.3.1 For every acyclic, well-behaved Standard Workflow Model that has only one final task and does not have a parallel exit from a parallel structure or a synchronised entry into a parallel structure there is an equivalent Structured Workflow Model.

For some well-behaved Standard Workflow Models that have more than one final task it is also possible to find an equivalent Structured Workflow Model. An example of such equivalent models is given in Figure 5.22.

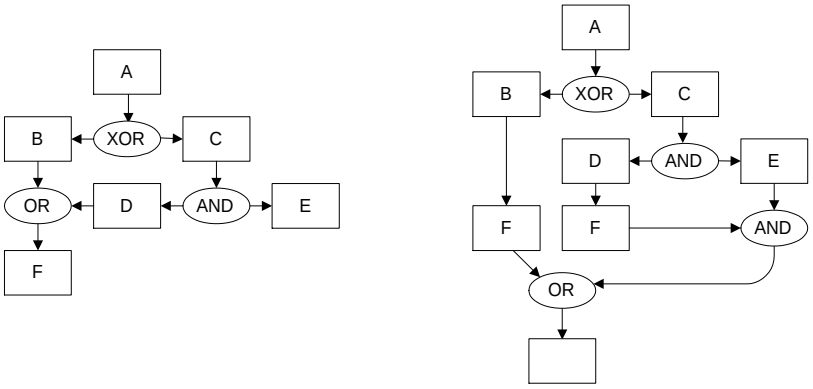

Figure 5.22: Transformation of a workflow with parallel exit from decision structure

Table 5.1 gives an overview of the main results of this section.

Pattern	**Transformation Possibility**
Entry into parallel structure	Transformation is possible for well-behaved models
Exit from parallel structure	No transformation possible as models result in deadlock
Synchronized entry into a decision	No transformation possible as models result in deadlock
Parallel exit from a decision	Transformation is possible for well-behaved models
Synchronized entry into a parallel structure	Transformation is typically impossible
Parallel exit from a parallel structure	Transformation is typically impossible
Entry into decision	Transformation is possible
Exit from decision	Transformation is possible

Table 5.1: Transformations of Structured Workflow Models with parallelism but without loops

5.3.3 Workflows with Parallelism and Loops

Finding out whether a workflow can deadlock or not in the context of loops is much more complex . To expose potential difficulties let us concentrate on what kind of loops we can encounter in a workflow model once AND-Join and AND-Split constructs are used. Every cycle in a graph has an entry point that can be either an OR-Join or an AND-Join and an exit point that can be either an AND-Split or a XOR-Split. Cycles without an entry point cannot start and cycles without an exit point cannot terminate. The latter case can be represented by a cycle with an exit point where the exit condition on the XOR-Split is set to false.

Most cycles will have OR-Joins and XOR-Splits as entry and exit points respectively (note that there may be many exit and entry points in the cycle) provided that the workflow is well-behaved. The transformation of such cycles is straightforward using transformations as presented earlier in this section.

If the cycle has an AND-Join as an entry point, the workflow will most likely deadlock. Examples of two workflows containing cycles with AND-Join as an entry-point that do not deadlock are shown in Figure 5.23 (note that these cycles can be seen as having OR-Joins as entry points and containing a synchronised entry into them).

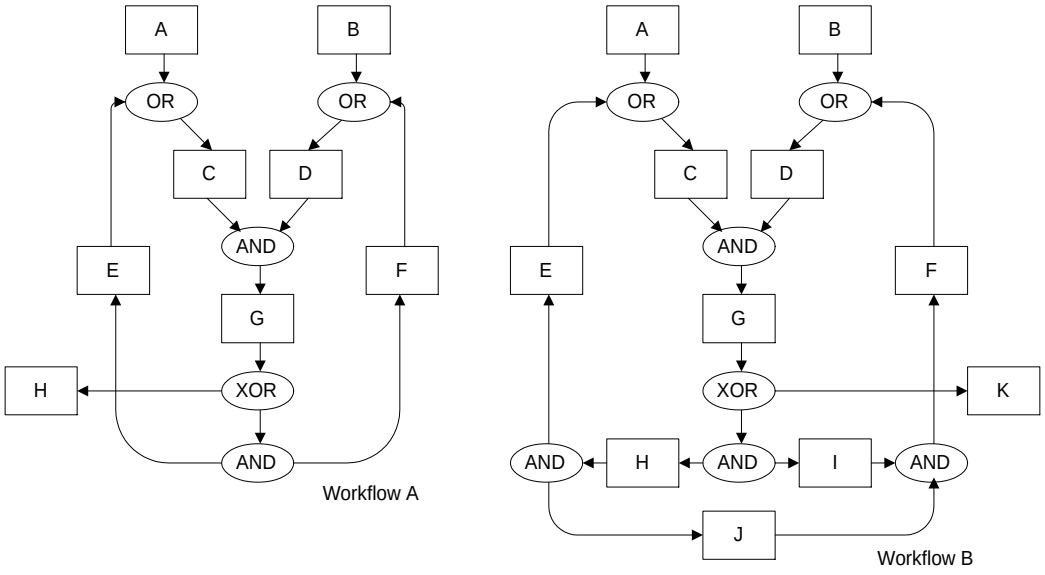

Figure 5.23: Two workflow models with arbitrary loops

Conversely, most workflows that have an AND-Split as an exit point will most likely result in multiple instances. Our previous observation that any workflow resulting in deadlock or multiple instances cannot be modelled as a structured workflow certainly

holds whether or not the workflow has loops. The major impact of introducing loops though is that finding out if the workflow deadlocks or results in multiple instances becomes a non-trivial task (see [HOR98, HO99][1]).

In rare cases when a cycle has an AND-Join as entry and an AND-Split as exit point and the workflow involved does not deadlock nor result in multiple instances, theorem 5.3.3 is helpful when determining if such a workflow can be remodelled as a structured workflow. In Figure 5.23 for example, workflow A can be remodelled as a structured workflow whereas we conjecture that workflow B cannot. The equivalent workflow to workflow A is shown in Figure 5.24.

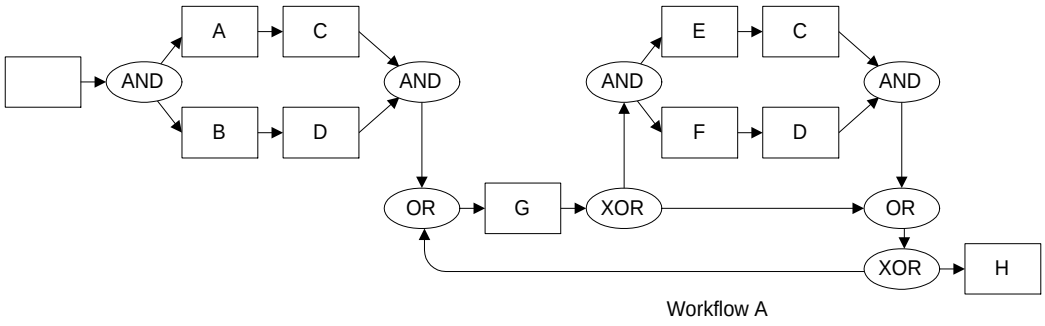

Figure 5.24: Structured version of leftmost workflow of Figure 5.23

5.4 Synchronizing Workflow Models

This section concentrates on a precise characterization of the expressive power of Synchronizing Workflow Models. To this end, we start with discussing some elementary properties.

First it is important to observe that arbitrary loops would cause problems in Synchronizing Workflow Models. Consider for example an activity A which is to trigger an activity B, while there is a trigger back from B to A. Activity A can only be executed if all its incoming triggers have been evaluated. However, one of these triggers depends on activity B, which on its turn depends on activity A resulting in immediate deadlock. For this reason only acyclic Synchronizing Workflow Models are considered in the remainder of this section.

Synchronizing Workflow Models have the property that every process element will receive exactly one token, true or false, for each of its input branches, and as a result

[1]These papers overlook the fact that Task Structures without decomposition are less expressive than Petri nets (as it is suggested otherwise) hence some complexity results need to be reconsidered.

it will produce a token for each of its outgoing branches. In Petri net terms this means that for every process element e of the model, exactly one of the corresponding transitions AT_e or AF_e will fire once. This result then effectively shows that Synchronizing Workflow Models are safe and never deadlock. Before the proof is presented let us first present some fundamental properties of Synchronizing Workflow Nets.

The following lemma can be proved by case distinction.

Lemma 5.4.1 Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ be a Synchronizing Workflow Model and $e \in \mathcal{P}$ be a process element of $\mathcal{W}$ that is enabled in a reachable marking M of the corresponding net system of $\mathcal{W}$, then:

1. There is always a transition of the associated net of e which is enabled;
2. Firing this transition results in a marking where e is completed.

While the above lemma provides a sufficient condition for at least one of the transitions associated with a process element to be enabled, the following lemma shows that this condition is also necessary (again the proof can be given using case distinction).

Lemma 5.4.2 Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ be a Synchronizing Workflow Model, $e \in \mathcal{P}$ a non-initial process element of $\mathcal{W}$ and $x \in in(e)$. Then for any marking M of the corresponding net of $\mathcal{W}$ such that $M(rt_{e,x}) = 0 \wedge M(rf_{e,x}) = 0$, none of the transitions in $T^e_{\mathcal{W}}$ is enabled.

Theorem 5.4.1 Let $\mathcal{W} = \langle \mathcal{P}, \mathsf{Trans}, \mathsf{Name} \rangle$ be a Synchronizing Workflow Model. Any process element $e \in \mathcal{P}$ in this model will fire exactly once.

Proof:

By induction over the depth n of process elements, where the depth of the process element is defined as the *longest* path from this process element to a process element without incoming transitions.

The case of $n = 0$ is obvious. Indeed, any process element with no incoming branches is initially enabled, and they will fire exactly once as they cannot be enabled again after they have fired.

For the induction step consider an arbitrary process element p at depth n. All its input elements have a depth less than n, hence it can be assumed that they will fire exactly once. According to Lemma 5.4.2, process element p cannot fire before all input elements have actually fired. Once this has happened, process element p is enabled (Lemma 5.4.1). As an enabled process element cannot be disabled by firing other process elements, process element p will indeed eventually fire. It cannot be re-enabled as its input elements will never fire again. Hence it can be concluded that process element p will fire exactly once. □

Corollary 5.4.1 Synchronizing Workflow Models are safe.

Corollary 5.4.2 Synchronizing Workflow Models do not have a deadlock.

Alternative proofs of these two corollaries were given in [HK99].

In the remainder of this section focus is on the expressive power of Synchronizing Workflows in relation to Standard Workflows. We will show that for any acyclic, well-behaved Standard Workflow Model there is an equivalent Synchronizing Workflow Model. Focus is on acyclic models as in our definition of Synchronizing Workflow Models cycles are not allowed and we have not made provision for the formal specification of iterative behaviour through decomposition[2]. Similarly only well-behaved models are considered as according to Theorem 5.4.1 Synchronizing Workflow Models never result in deadlock and are always safe.

Definition 5.4.1
A WB-system *is a labelled Petri net system which corresponds to an acyclic, well-behaved Standard Workflow Model.* □

The following proposition captures the formal properties of WB-systems (which can easily be verified).

Proposition 5.4.1 A WB-system $\mathcal{P} = \langle P, T, F, L, M_0 \rangle$ has the following properties:

- There are no sink places (i.e. a place p such that $p\bullet = \varnothing$);
- The net is free-choice;
- Every node $x \in P \cup T$ is on a path from a source place (i.e. a place p such that $\bullet p = \varnothing$);
- The net is safe starting from the initial marking with just tokens in source places;
- There are no dead transitions starting from the initial marking with just tokens in source places;
- From any marking reachable from the initial marking with just tokens in source places, it is possible to reach the empty marking.

[2]Iterative behaviour in Synchronizing Workflow Models is typically realized through decomposition and arguments related to such a solution are similar to those presented in the discussion on loops in Structured Workflow Models.

The results that follow summarise some important characteristics of WB-systems. These will be useful for providing a formal relationship between acyclic well-behaved Standard Workflow Models and Synchronizing Workflow Models.

Definition 5.4.2

Let $\mathcal{P}$ be a WB-system and $S_{\mathcal{P}}$ the set of its sink transitions, i.e. $S_{\mathcal{P}} = \{t \in T \mid t\bullet = \varnothing\}$. For any $s \in S_{\mathcal{P}}$, P^s is the set of places from which s is reachable by following the arcs in F, i.e. for each place $p \in P^s$ there is a directed path from p to s, and T^s is the set of transitions which consumes tokens from P^s but does not produce any token for P^s. □

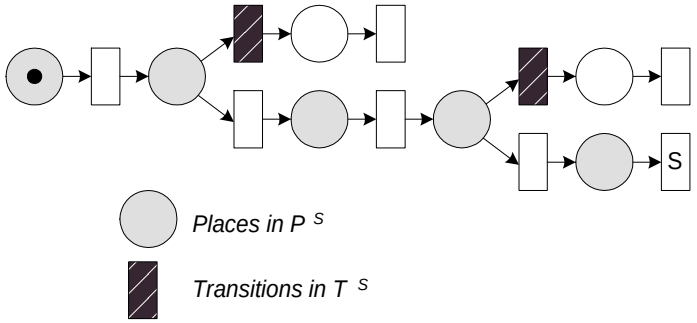

Figure 5.25: Example of sets P^s and T^s

Example 5.4.1 A simple example of the above definition is depicted in Figure 5.25. □

Lemma 5.4.3 Let $\mathcal{P}$ be a WB-system. Whenever a place $p \in P^s$ is marked, s can fire, i.e. there is a firing sequence enabling s.

Proof:

Let M be a marking that marks place p. If $p \in \bullet s$ then the lemma holds since, as the net does not deadlock and is free-choice it is always possible to fire transition s. Let $x =< p_1, t_1, ..., p_n, t_n >$ be a directed path with $p_1 = p$ and $t_n = s$. As the net does not deadlock and is free-choice, there must be a firing sequence that enables transition t_1. Firing t_1 marks place p_2. By recursively applying the argument to the remaining places and transitions it is possible to construct a firing sequence σ such that $M \xrightarrow{\sigma} M'$, and M' is a marking that marks a place q such that $q \in \bullet s$. □

Lemma 5.4.4 Let $\mathcal{P}$ be a WB-system. Transitions in $S_{\mathcal{P}}$ can fire only once.

Proof:

If a sink transition can fire twice, it is possible to delay the first firing until the second one and clearly the WB-system is not safe in that case. □

Lemma 5.4.5 Let $\mathcal{P}$ be a WB-system. Firing a transition from T^s permanently disables s.

Proof:

This lemma is the most complex one. To prove this it is shown that the places in P^s become unmarked after firing a transition in T^s. Consider a place $p_1 \in P^s$ which contains a token which can be removed by firing a transition t in T^s and another place $p_2 \in P^s$ which remains marked after firing t. Suppose that t fires, then, based on Lemma 5.4.3, there is a firing sequence enabling s. If t does not fire, the same firing sequence is enabled. However, this implies that after executing this sequence, p_1 is still marked, and based on Lemma 5.4.3, s could fire again. This is not possible as indicated by Lemma 5.4.4. Therefore, all places in P^s become unmarked after firing a transition in T^s. □

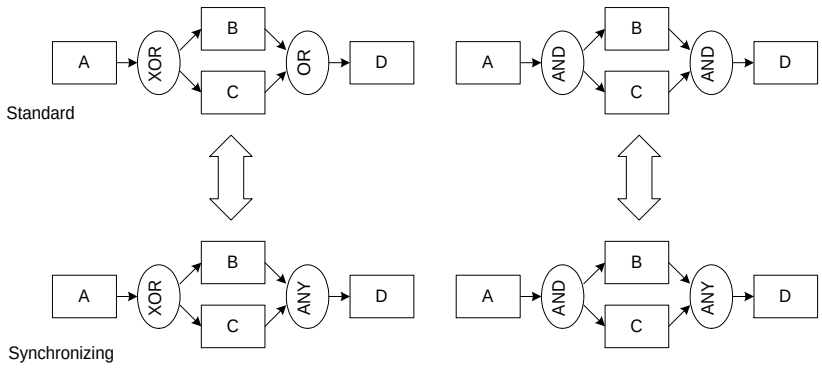

Figure 5.26: Equivalent Standard and Synchronizing Workflows

In order to examine the expressive power of Synchronizing Workflow Models, it is important to fully understand the expressive power of its ANY-Join construct. To this end, consider the workflow depicted in Figure 5.26. It is easy to see that the workflows on the left are bisimulation equivalent, as are the workflows on the right. Note that both the OR-Join and the AND-Join have the ANY-Join as their equivalent. Given the fundamentally different semantics of the OR-Join and the AND-Join in Standard Workflows this may come as a surprise. It can even be taken further in the sense that replacing all OR-Joins and AND-Joins in any acyclic well-behaved Standard Workflow Model with ANY-Joins as well as replacing all other constructs in Standard

Workflows with their equivalent representations in Synchronizing Workflows results in an equivalent model (formally captured in Theorem 5.4.2). This provides a first indication of the expressive power of Synchronizing Workflows.

Definition 5.4.3

Let $\mathcal{W} = \langle \mathcal{P}, \mathcal{J}_o, \mathcal{J}_a, \mathcal{S}_o, \mathcal{S}_a, \mathcal{A}, \mathsf{Trans}, \mathsf{Name} \rangle$ *be an acyclic well-behaved Standard Workflow Model. The* corresponding *Synchronizing Workflow Model* $\mathcal{S} = \langle \mathcal{P}, \mathcal{J}_o, \mathcal{J}_a, \mathcal{S}_o, \mathcal{S}_a, \mathcal{A}, \mathsf{Trans}, \mathsf{Name} \rangle$ *is defined by:*

$$
\begin{array}{lcll}
A^{\mathcal{S}} & = & A^{\mathcal{W}} & \#\ \textit{same activities}\ \# \\
S_o^{\mathcal{S}} & = & S_o^{\mathcal{W}} & \#\ \textit{same XOR-Splits}\ \# \\
S_a^{\mathcal{S}} & = & S_a^{\mathcal{W}} & \#\ \textit{same XOR-Splits}\ \# \\
J_o^{\mathcal{S}} & = & J_a^{\mathcal{W}} \cup J_o^{\mathcal{W}} & \#\ \textit{ANY-Joins for each of the OR-Joins \& AND-Joins}\# \\
J_a^{\mathcal{S}} & = & \varnothing & \#\ \textit{no ALL-Joins}\# \\
\mathsf{Trans}^{\mathcal{S}} & = & \mathsf{Trans}^{\mathcal{W}} & \#\ \textit{same transitions}\# \\
\mathsf{Name}^{\mathcal{S}} & = & \mathsf{Name}^{\mathcal{W}} & \#\ \textit{same labeling}\#
\end{array}
$$

□

The next step is to show that for any Standard Workflow Model the corresponding Synchronizing Workflow Model is indeed bisimulation equivalent. This is complex and requires some preparation.

First an essential property of well-behaved Standard Workflow Models is formally captured. This is the fact that in any reachable marking of a Standard Workflow net for any marked place, there is no other marked place on a path from an initial place to that marked place.

Proposition 5.4.2 Let $\mathcal{W}$ be an acyclic, well-behaved Standard Workflow Model, $(PN_{\mathcal{W}}, M_0)$ its corresponding net system and let $x =< p_1, t_1, p_2, t_2, ..., t_{n-1}, p_n >$ with $p_1 = p$ and $p_n = q$ be an acyclic directed path. For any reachable marking M we have $M(q) = 1 \Rightarrow M(p) = 0$.

Proof:

If p were marked, another token can be produced for q according to Lemma 5.4.3. Hence the net would not be safe. Contradiction. □

A similar result holds for Synchronizing Workflow Models, except that a distinction needs to be made between true places and false places.

Proposition 5.4.3 Let $\mathcal{W}$ be a Synchronizing Workflow Model, $(PN_{\mathcal{W}}, M_0)$ its corresponding net system and M a reachable marking of $(PN_{\mathcal{W}}, M_0)$. Let p be a true place and $\overline{p}$ its corresponding false place, and q another true place and $\overline{q}$ its corresponding false place such that there is a direct, acyclic path from p to either q or $\overline{q}$ then

$$(M(q) = 1 \vee M(\overline{q}) = 1) \Rightarrow (M(p) = 0 \wedge M(\overline{p}) = 0).$$

Proof:

In Synchronizing Workflow Model $\mathcal{W}$, if the place p or $\overline{p}$ contains a token, there is a firing sequence producing a token for either place q or $\overline{q}$ (Theorem 5.4.1). If one of these places already has a token (suppose it is a true place), according to the Monotonicity Lemma (see e.g. p.22 of [DE95]) through application of this firing sequence a second token can be produced for this place or its corresponding false place. □

Having established some basic properties of well-behaved Standard Workflows and Synchronizing Workflows, it is possible to show that any Synchronizing net can be simulated by a WB-system and vice versa, thus demonstrating that they are *simulation equivalent*. Having achieved this, it is possible to give a bisimulation relation, thus proving that they are in fact bisimulation equivalent.

The main difficulty in simulating a Standard Workflow Model by a Synchronizing Workflow Model is that the latter essentially propagates two types of tokens. For every firing of a process element of a Standard Workflow Model (propagation of a true token) we may need to fire a number of process elements in the corresponding Synchronizing Workflow Model in order to propagate some false tokens. Such a need typically arises when we want to fire an OR-Join in a Standard Workflow Model . The corresponding ANY-Join in the Synchronizing Workflow Model requires tokens for each of its inputs, hence some false tokens may need to be propagated. Lemma 5.4.6 guarantees that this is always possible. First however it is necessary to define the notion of workflow instances being *true-token-equivalent* which informally equates a marking of a Standard Workflow Model $\mathcal{W}$ with a marking of the corresponding Synchronizing Workflow Model $\mathcal{S}$ if for every token in the associated net of a process element of $\mathcal{W}$ there is a token in the true place of the associated net of the corresponding process element of $\mathcal{S}$.

Definition 5.4.4

Let $\mathcal{W}$ be a Standard Workflow Model and $\mathcal{S}$ its corresponding Synchronizing Workflow Model and let $PN_{\mathcal{W}}$ and $PN_{\mathcal{S}}$ be the corresponding Petri net systems of these models respectively. Let M_1 be a reachable marking of $PN_{\mathcal{W}}$. A reachable

marking M_2 of $PN_{\mathcal{S}}$ is said to be true-token-equivalent *with M_1 iff for all places $p \in \mathsf{True}^{\mathcal{S}}, M_2(p) = 1 \iff M_1(h(p)) = 1$, where h is an injection from $\mathsf{True}^{\mathcal{S}}$ to the corresponding places in $PN_{\mathcal{W}}$, i.e. $h(rt_{x,y}) = r_{x,y}, h(ct_{x,y}) = c_{x,y}$ and $h(rt_x) = r_x$.* □

Lemma 5.4.6 Let $\mathcal{W}$ be a Standard Workflow Model and $\mathcal{S}$ its corresponding Synchronizing Workflow Model and let $PN_{\mathcal{W}}$ and $PN_{\mathcal{S}}$ be the corresponding Petri net systems of these models respectively. Let M_1 be a reachable marking of $PN_{\mathcal{W}}$ and M_2 a true-token equivalent marking of $PN_{\mathcal{S}}$, then there exists a (possibly empty) firing sequence $\sigma = t_1 t_2 ... t_n$ of λ-transitions in the Synchronizing Workflow Model such that $M_2 \xrightarrow{\sigma} M_2'$ where M_2' is a marking such that for every enabled OR-Join in $\mathcal{W}$ the corresponding ANY-Join in $\mathcal{S}$ is enabled.

Proof:

Without loss of generality we can focus on an enabled OR-Join with two incoming transitions in a marking M_1. As the Standard Workflow net is safe, only one ready place of the associate net of this OR-Join can hold a token (and one token only). As the marking M_2 in the corresponding Synchronizing Net is true-token-equivalent, the corresponding true place of the associated net of the ANY-Join will hold a token. According to Theorem 5.4.1, a token will have to arrive for the other branch of the ANY-Join. More formally, let p be the true ready place of this branch of the associated net of this ANY-Join. Then there is a marking M such that p or $\overline{p}$ is marked.

In the Standard Workflow net, according to Proposition 5.4.2, there cannot be a token on a path from an initial activity to the OR-Join (otherwise the OR-Join could fire twice). As M_1 and M_2 are true-token-equivalent, in M_2 there cannot be a token in a true place on any path from an initial activity of $\mathcal{S}$ to either p or $\overline{p}$. Hence on a path from an initial activity to place p or $\overline{p}$ there must be a false place that contains a token (there can be more than one such token). Moving such tokens involves the firing of λ-transitions only (note that there cannot be ANY-Joins on such a path with true-tokens waiting, as that would mean that in the Standard Workflow net there is a token for the corresponding OR-Join in contradiction to Proposition 5.4.2). Note that firing these transitions will result in marking in which $\overline{p}$ is marked (p cannot be marked). □

Lemma 5.4.7 Let $\mathcal{W}$ be an acyclic well-behaved Standard Workflow Model, $(PN_{\mathcal{W}}, M_0)$ its corresponding net system, $\mathcal{S}$ its corresponding Synchronizing Workflow Model and $(PN_{\mathcal{S}}, M_0)$ its net system, then $PN_{\mathcal{W}}$ and $PN_{\mathcal{S}}$ are simulation equivalent.

Proof:

First focus is on simulating the Synchronizing Workflow Model by the Standard Workflow Model. This is achieved through induction on the number n of firings of process elements. The case of $n = 0$ follows from the fact that the initial markings of these workflow models are true-token-equivalent. Assume that after firing n process elements, marking M_1 of the Synchronizing Workflow Net and marking M_2 of the Standard Workflow Net are true-token-equivalent. We then fire a process element p in the Synchronizing Workflow Model which results in marking M_1'.

We will show that either M_1' and M_2 are true-token-equivalent or it is possible to fire a corresponding element of the Standard Workflow Model and the resulting marking M_2' and M_2 are true-token-equivalent.

This is achieved through a straightforward case distinction:

1. If p was enabled with only false tokens, firing it resulted in a marking that is true-token-equivalent to M_2. No action needs to be performed in the Standard Workflow Model. From this moment on we assume that at least one of the enabling tokens of p was a true token.
2. If p is an activity, the corresponding activity in the Standard Workflow Model can be performed.
3. If p is a Split, the corresponding Split in the Standard Workflow Model can be performed, and the resulting marking is again true-token-equivalent.
4. If p is an ANY-Join with more than one true token, one can conclude that the corresponding Join in the Standard Workflow Model has to be an AND-Join, as otherwise this workflow would not be safe. Firing this AND-Join results again in true-token-equivalent markings.
5. If p is an ANY-Join with one true and the rest false tokens, one can conclude that the corresponding Join in the Standard Workflow Model has to be an OR-Join, as otherwise there would be a deadlock (as according to Proposition 5.4.3 there are no tokens in places above). Firing this OR-Join results again in true-token-equivalent markings.

The opposite, simulating the Standard Workflow Model by the corresponding Synchronizing Workflow Model, uses a similar case distinction. The only real problem is that Lemma 5.4.6 is needed in order to guarantee that if an OR-Join is performed in the Standard Workflow net, the corresponding ANY-Join in the Synchronizing Workflow Model can be performed. □

Corollary 5.4.3 Let $\mathcal{W}$ be an acyclic well-behaved Standard Workflow Model, $PN_{\mathcal{W}}$ its corresponding net, $\mathcal{S}$ its corresponding Synchronizing Workflow Model and

$PN_{\mathcal{S}}$ its net, then for every reachable marking M of $PN_{\mathcal{W}}$ there is a reachable marking M' of $PN_{\mathcal{S}}$ which is true-token-equivalent to M. Similarly for every reachable marking M of $PN_{\mathcal{S}}$ there is a reachable marking M' of $PN_{\mathcal{W}}$ true-token-equivalent to M.

Theorem 5.4.2 Let $\mathcal{W}$ be an acyclic well-behaved Standard Workflow Model, $PN_{\mathcal{W}}$ its corresponding net, $\mathcal{S}$ its corresponding Synchronizing Workflow Model and $PN_{\mathcal{S}}$ its net, then $PN_{\mathcal{W}}$ and $PN_{\mathcal{S}}$ are bisimulation equivalent.

Proof:

Before defining a bisimulation relation, we introduce labels for all transitions, except those that just propagate false tokens. The reason for this is that we would like the bisimulation relation to maintain the relationship between the execution of the various corresponding joins and splits in the two nets. Naturally, by showing that the resulting nets are equivalent, it follows that the original nets with fewer labels, are also equivalent. The bisimulation relation is just made a bit more strict.

We now define a relation R between the reachable markings of both nets and show that it is a bisimulation relation. Formally, R relates two markings if and only if they are true-token-equivalent. From Corollary 5.4.3 it then follows that for every reachable marking M_1 of the Standard Workflow net, there is a reachable marking M_2 of the Synchronizing Workflow net such that $(M_1, M_2) \in R$ and vice versa.

Let $(M_1, M_2) \in R$. First it will be shown that for every label a and marking M_1' of the Standard Workflow net such that $M_1 \stackrel{a}{\Longrightarrow} M_1'$ there is a marking M_2' in the Synchronizing Workflow net with $M_2 \stackrel{a}{\Longrightarrow} M_2'$ and $(M_1', M_2') \in R$. This requires the following case distinction:

1. If the label corresponds to an activity, then the corresponding activity can be performed in the Synchronizing Workflow net.
2. If the label corresponds to a split, then the corresponding split can be performed in the Synchronizing Workflow net.
3. If the label corresponds to an AND-Join then the corresponding join in the Synchronizing Workflow net can be performed, as all input branches will have true tokens.
4. The case where the label corresponds to an OR-Join is the most interesting one. In the Synchronizing Workflow net, the corresponding join will have exactly one true token. If all other branches have false tokens, then the join can be performed directly. If not, then according to Lemma 5.4.6, false tokens can be propagated using λ-transitions only.

Note that in all the above cases, the resulting markings are true-token-equivalent to the resulting marking in the Standard Workflow net, hence related in R.

Now it will be shown that for every label a and marking M_2' of the Synchronizing Workflow net such that $M_2 \stackrel{a}{\Longrightarrow} M_2'$ there is a marking M_1' in the Standard Workflow net with $M_1 \stackrel{a}{\Longrightarrow} M_1'$ and $(M_1', M_2') \in R$. This requires a similar case distinction (note that we do not need to consider transitions propagating false tokens, as such transitions are unlabelled):

1. If the label corresponds to an activity, then the corresponding activity can be performed in the Standard Workflow net.
2. If the label corresponds to a split, then the corresponding split can be performed in the Standard Workflow net.
3. If the label corresponds to an ANY-Join with more than one true token, then this join corresponds to an AND-Join in the Standard Workflow net (otherwise the workflow would not be safe). This AND-Join can be performed as all its input branches will have tokens.
4. If the label corresponds to an ANY-Join with one true token and the rest false tokens, then this join corresponds to an OR-Join in the Standard Workflow net (otherwise deadlock would occur). Again, this OR-Join can then fire.

Note that in all the above cases the resulting markings are true-token-equivalent to the resulting marking in the Synchronizing Workflow net, hence related in R. Therefore, and given that R relates the initial markings of both systems, R is a bisimulation relation. □

Theorem 5.4.2 has important practical ramifications, as it effectively demonstrates that the choice for a true/false token evaluation strategy when developing a workflow engine does not compromise the expressive power of the workflow language involved as long as well-behaved workflows with structured loops only are considered. One advantage of this approach is that workflow analysts need not worry about deadlock, as all their specifications are guaranteed to be deadlock free.

Having established which Standard Workflow Models can be captured as Synchronizing Workflow Models, one may wonder whether all Synchronizing Workflow Models have a Standard Workflow equivalent. Intuitively, the fact that the Petri net representation of both ANY-Join and ALL-Join is non-free-choice hints at the possibility that Synchronizing Workflow Models may exist which do *not* have a Standard Workflow equivalent. The next theorem proves this fact formally.

Theorem 5.4.3 There exist Synchronizing Workflow Models for which no Standard Workflow Model can be found such that the corresponding Petri nets are bisimulation equivalent.

Proof:

Consider the Synchronizing Workflow Model $\mathcal{W}$ of Figure 5.27. We will show that no free-choice net system exists that is equivalent to this workflow. This then concludes the proof as the corresponding net system of any Standard Workflow Model is free-choice.

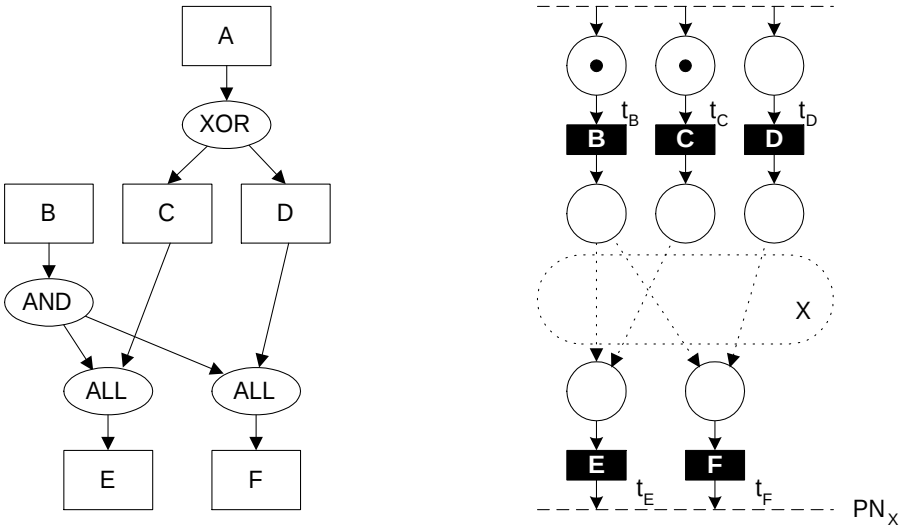

Figure 5.27: ALL-Join adds expressive power

Let $PN_{\mathcal{W}}$ be the corresponding net of Synchronizing Workflow Model $\mathcal{W}$. Let $\mathcal{X}$ be a Standard Workflow Model that is equivalent to $\mathcal{W}$ and PN_x its corresponding net.

Playing the bisimulation game we have that in PN_x there must be a reachable marking M_1 that enables a transition labelled A and a transition labelled B and does not enable any transitions labelled C, D, E or F. Let us refer to the enabled transitions as t_A and t_B respectively.

For PN_x to be bisimulation equivalent to $PN_{\mathcal{W}}$ there must be a marking M_2 such that $M_1 \stackrel{a}{\Longrightarrow} M_2$ and M_2 is a marking that enables t_B and a transition labelled C and does not enable any transitions labelled A, D, E or F. Let us refer to the enabled transition labelled C in M_2 as t_C.

Similarly must be a marking M_3 such that $M_1 \stackrel{a}{\Longrightarrow} M_3$ and M_3 is a marking that enables t_B and a transition labelled D and does not enable any transitions

labelled A, C, E or F. Let us refer to the enabled transition labelled D in M_3 as t_D.

The bisimulation game further yields that there must be a marking M_4 such that $M_2 \stackrel{bc}{\Longrightarrow} M_4$ and M_4 is a marking that enables a transition labelled E and does not enable any other labelled transitions. Let us refer to the enabled transition labelled E in M_4 as t_E.

Similarly there must be a marking M_5 such that $M_2 \stackrel{bd}{\Longrightarrow} M_5$ and M_5 is a marking that enables a transition labelled F and does not enable any other labelled transitions. Let us refer to the enabled transition labelled F in M_5 as t_F.

As PN_x is the corresponding net of a Standard Workflow Model, transitions t_B, t_C, t_D, t_E and t_F have exactly one input and one output place. A subnet of PN_x along with transitions t_B, t_C, t_D, t_E and t_F with marking M_2 is schematically shown as the right diagram of Figure 5.27. The subnet comprising output places of transitions t_B, t_C and t_D, input places of transitions t_E, t_F and subnet X of Figure 5.27 fullfils the assumptions of Lemma 5.2.2 (Selective Synchronizer) hence PN_x cannot be free-choice which contradicts the assumption that PN_x is the corresponding net of a Standard Workflow Model. □

Summarizing, as opposed to Standard Workflow Models, Synchronizing Workflow Models are always safe, they never result in deadlock, and they do not allow for direct specification of arbitrary cycles. Synchronizing Workflow Models can express all Standard Workflow Models that do have these properties (i.e. well-behaved, acyclic models). There are Synchronizing Workflow Models though that are inherently non free-choice and hence do not have a Standard Workflow equivalent.

5.5 Summary

In this Chapter we have presented some fundamental results concerning theoretical expressiveness limits of the four language classes introduced in Chapter 4. The following is a summary of the results with practical implications for the workflow modeller:

1. Standard Workflow Models are less expressive than free-choice Petri nets. This is a serious expressiveness limitation of models comprising activities, OR-Joins, AND-Joins, XOR-Splits and AND-Splits. There is no good solution to overcome this problem though. One approach is to base the modelling of workflow process entirely on Petri nets. However, the resulting model is much more complex and

may not be suitable for end-users (examples of commercial workflow management systems based on Petri nets are COSA [SL99] and Income [Pro98]). The common alternative for certain scenarios is to use data flow as an augmentation to control flow. This is discussed in more details in section 6.5. Another approach is to enhance the modelling language with some additional constructs. The downside is that there is no consensus in industry what advanced constructs should be added to the basic set of process modelling elements thus making model interchange between different products hard or impossible. Some of these advanced constructs and their relative expressive power will be presented in Chapter 6.

2. Safe Workflow Models are less expressive than Standard Workflow Models. This relates to the fact that Standard Workflow Models can be unbounded as well as the fact that it is possible to use an AND-Join construct in a way beyond the expressive power of Safe Workflow Models. We believe there is no reason to restrict the power of a workflow modelling language to safe models only.

3. Structured Workflow Models are less expressive than Safe Workflow Models. Transformations of arbitrary models are hard or impossible. Again, we believe that there is no reason to restrict the power of a workflow modelling language to structured models only.

4. Synchronizing Workflow Models and Standard Workflow Models are incomparable. However, intuitively it seems that Synchronized Models are far more restrictive than Standard Workflow Models. First of all with Synchronizing Workflow Models it is awkward to model loops and decomposition needs to be used. Secondly it is not possible to model multiple instances which is a serious limitation. Some extra expressive power of Synchronizing Models does not seem to offset their obvious limitations and this can perhaps be overcome by augmenting Standard Workflow Models with some additional advanced modelling construct.

We believe that we have provided strong arguments for the choice of Standard Workflow modelling evaluation strategy over alternative workflow evaluation approaches. It is therefore most surprising that this evaluation approach does not seem to be a common choice amongst workflow vendors. As much as we cannot deduct the exact reasoning behind the choice of a particular approach by a given vendor, we believe that there may be two primary reasons for that:

- The attempt to keep the modelling language simple and, in particular, to prevent workflow models from having undesirable deadlocks and multiple instances. There are some efforts in academia (e.g. [SO99, AHV97]) to derive verification

engines that could analyse workflow models for possible errors before they are deployed. We are unaware, however, of any commercial vendor utilizing such a verification engine.

- The lack of anticipation for certain business requirements, in particular for the need for multiple instances. This seems to be particularly true for products that have chosen the Safe Workflow Modelling approach. Indeed, support for multiple instances requires more complex engineering efforts and that could have been overlooked in the early years of workflow technology.

Chapter 6

Advanced Expressiveness Results

The different evaluation strategies and the semantics of the basic control flow constructs are not the only areas not precisely addressed by the WfMC. In this Chapter we would like to investigate some other issues associated with choices that workflow engine designers are likely to face.

When defining syntax and semantics for workflow models, it was assumed that there may be multiple final activities in a workflow model. There are some workflow engines (e.g. Verve Workflow, Forté Conductor, etc.) for which this assumption does not hold. In Section 6.1 the consequences associated with this design decision are explored.

The execution of Standard Workflow Models (as opposed to Synchronizing and Structured Workflow Models) may result in deadlock. Typically this is viewed as an undesirable situation. In Section 6.2 we consider the possibility of using deadlock intentionally to express certain task dependencies and determine whether this can enhance the expressive power of Standard Workflow Models.

Some workflow languages support constructs, not part of the basic control flow constructs, which clearly have practical significance. One such construct is considered in Section 6.3, where it is shown that it cannot be simulated using the basic control flow constructs.

In [Wor99b] WfMC defines a subprocess as a "process that is enacted or called from another (initiating) process (or sub process), and which forms part of the overall (initiating) process". Similarly to definitions of other control flow constructs this definition leaves room for different interpretations. In Section 6.4, decomposition and some of its many uses, are studied in more depth.

Finally in Section 6.5 we provide some insight into the use of data flow for augmenting control flow specifications in workflow models.

6.1 Termination

Termination refers to the state where no work remains to be done. Often, this situation is referred to as successful termination to distinguish it from deadlock. While the presented definition of termination in Section 4.1.1 seem straightforward, and languages supporting the synchronizing evaluation strategy employ it (e.g. MQSeries Workflow), most workflow engines in practice, especially those supporting standard or safe workflows (a notable exception here is Staffware), have a different view on termination. In these engines, for every workflow, one or more final tasks need to be specified. The workflow then is considered to be terminated when the first of these final tasks have completed.

This termination policy is particularly problematic when a (or the) final task is reached while some other parallel threads are still running. What the workflow engine will do in such a situation differs from product to product but typically the remaining threads are abruptly aborted leaving the workflow in a potentially inconsistent state. Hence we are interested in workflows where this situation cannot occur and we will refer to them as terminating *strictly*.

Definition 6.1.1 (*strictly terminating workflows*)
Let $(PN_{\mathcal{W}}, M_0)$ be the corresponding system of Standard Workflow Model $\mathcal{W}$. We will call $\mathcal{W}$ terminating strictly *iff for every sink transition t and every reachable marking M of $PN_{\mathcal{W}}$ that enables t, we have for all places p:*

$$M(p) = \begin{cases} 1 & \text{if } p \in {\bullet t} \\ 0 & \text{if } p \notin {\bullet t} \end{cases}$$

□

Definition 6.1.2 (*uniquely terminating workflows*)
A Standard Workflow Model $\mathcal{W}$ is terminating uniquely *iff it has exactly one final task and is terminating strictly.* □

Clearly, there exist non-safe Standard Workflow models for which there is no strictly terminating equivalent workflow model. A simple example of such a model is shown in Figure 6.1 and the practical usefulness of a relaxed termination strategy is evident when considering patterns involving multiple instances (see Section 3.4). Standard Workflow models that are well-behaved, on the other hand, have a terminating uniquely equivalent workflow model.

To prove this result we will show that for every well-behaved Standard Workflow model $\mathcal{W}$, it is possible to transform its corresponding Petri net, PN into a bisimulation

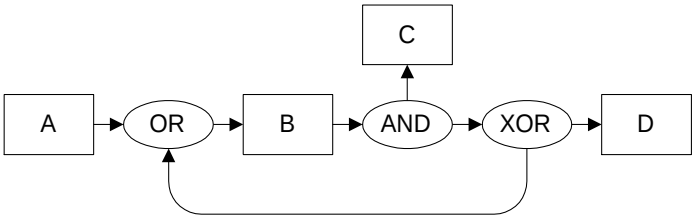

Figure 6.1: Sample Standard Workflow Model utilising relaxed termination policy

equivalent net PN' that has only one sink transition (i.e. a transition without output places). It is then possible to convert PN' back to a terminating uniquely workflow specification that is equivalent to $\mathcal{W}$.

Theorem 6.1.1 Every well-behaved Standard Workflow Model has an equivalent workflow model that is terminating uniquely.

Proof:

Let $\mathcal{W}$ be a well-behaved Standard Workflow model and $PN = \langle P, T, F \rangle$ be its corresponding WB-net with S_{PN}, P^s and T^s as defined in Definition 5.4.2. Then,

$$\begin{aligned} PN' \;=\; & (P \cup \{p_s | s \in S_{PN}\}, \\ & T \cup \{t_f\}, \\ & F \cup \{(t, p_s) | s \in S_{PN} \wedge t \in T^s\} \cup \{(p_s, t_f) | s \in S_{PN}\}) \end{aligned}$$

is a WB-net with one sink transition t_f.

Clearly, t_f is a sink transition. There are no other sink transitions because all (former) sink transitions in S_{PN} have an output place in $\{p_s | s \in S_{PN}\}$ (Note that $s \in T^s$ and $s\bullet = \{p_s\}$). Moreover, the source places of PN are still the only source places of PN', PN' is free-choice, and has no sink places (all new places have an input and output transition). It is also easy to see that every node is on a path from a source place. To prove the last three properties stated in Proposition 5.4.1, we show that from any reachable state it is possible to reach the empty state, i.e., enable and fire t_f.

Consider a (former) sink transition $s \in S_{PN}$. As long as P^s contains tokens, there is a firing sequence enabling s (Lemma 5.4.3). If a transition in T^s fires, the last token in P^s is consumed. Moreover, s can fire only once (Lemma 5.4.4). Note that $s \in T^s$ and exactly one source place is in P^s. On the one hand, only one transition of T^s can fire. Therefore, it is not possible to mark p_s more than once. On the other hand, at least one of the transitions of T^s will fire (assuming

fairness: initially the unique source place in P^s is marked and it is possible to reach the empty marking). Therefore, p_s will be marked at least once. Hence, the place p_s is marked once. Therefore, all places in $\{p_s | s \in S_{PN}\}$ are marked once and sink transition t_f will produce the empty marking.

As the resulting PN' net is an FCDA-net (as per Definition 5.1.1) it is straightforward to transform PN' back to a workflow model using transformations presented in Theorem 5.1.2.

It remains to be shown that PN' is bisimulation equivalent to PN. Let M be a reachable marking of PN. We will call a reachable marking M' of PN' an associated marking of M iff $M'[P] = M$ (in other words it should have exactly the same number of tokens in every place of the original net, the markings of the introduced places does not matter). From the construction of PN' it is easy to check that a relation R that relates every marking M of PN to all its associated markings in PN' is indeed a bisimulation relation.

□

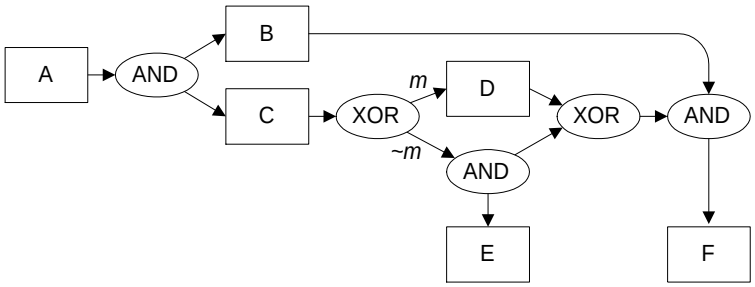

Figure 6.2: Sample Standard Workflow model with two final tasks

Example 6.1.1 As an example of the construction used in the proof of Theorem 6.1.1, consider the workflow of Figure 6.2. This workflow has two final tasks, named E and F. In every instance, the task named F is executed while the task named E is executed only if condition m evaluates to false. By following the construction presented in the proof we end up with the workflow presented in Figure 6.3. Note that this workflow is indeed equivalent to the one of Figure 6.2. Also note that from a comprehensibility point of view, the workflow with the unique final task is much more complicated and its control flow would be much harder to understand for an end-user. □

Remark 6.1.1

Naturally, the equivalent of Theorem 6.1.1 for Synchronizing Workflow models is

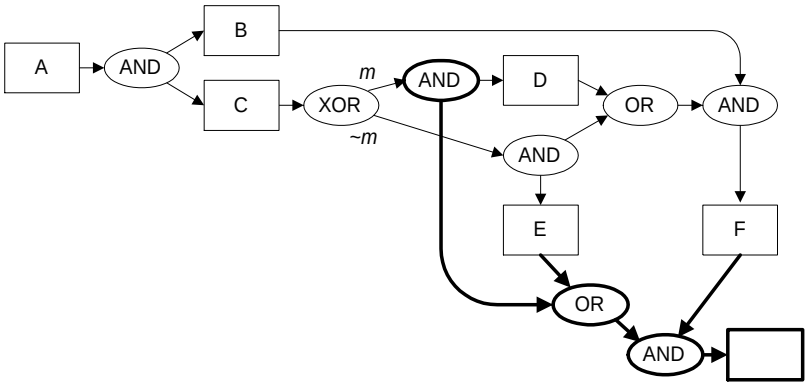

Figure 6.3: Terminating uniquely equivalent workflow to workflow of Figure 6.2

trivial, as for every Synchronizing Workflow Model $\mathcal{W}$ with more than one final task, the Synchronizing Workflow Model which simply adds an ANY-Join with input transitions from all the final tasks followed by a null activity is equivalent to $\mathcal{W}$. □

6.2 Deadlock

This section takes a closer look at the issue of deadlock in workflows. As Synchronizing Workflow models cannot deadlock, focus is on Standard Workflows exclusively.

Imagine a workflow management system that has the ability to detect deadlock at runtime (from a programming point of view this is fairly easy to achieve). Moreover, imagine that the workflow analyst could instruct the workflow engine what to do when it encounters a deadlock. Specifically, (s)he could instruct the engine to treat deadlock as a normal, successful, termination[1]. The question that we would like to address is whether such a feature would increase the expressive power of a workflow engine. More formally, this question boils down to determining whether any Standard Workflow model with a deadlock has an "equivalent" deadlock free Standard Workflow model. As our equivalence notion will always distinguish a specification that deadlocks from a specification that does not deadlock, a relaxed equivalence notion is required.

Definition 6.2.1

Workflow models $\mathcal{W}_1$ and $\mathcal{W}_2$ are execution equivalent *iff the begin-end transformations of their corresponding systems are bisimilar according to Definition 4.2.3 excluding the second clause.*

□

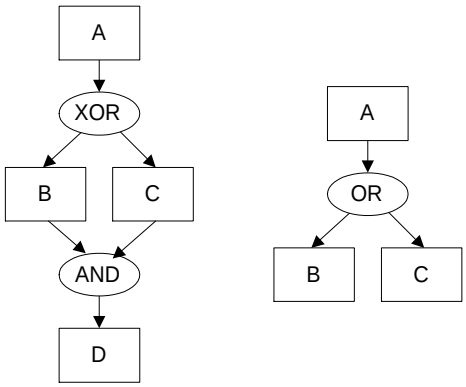

Figure 6.4: Two execution equivalent processes

Example 6.2.1 The two workflow processes depicted in Figure 6.4 are execution equivalent even though the left-most process deadlocks whilst the right-most process always terminates successfully. □

Theorem 6.2.1 (*dynamic deadlock resolution adds expressive power*) There exist Standard Workflow models for which no deadlock free execution equivalent Standard Workflow model exists.

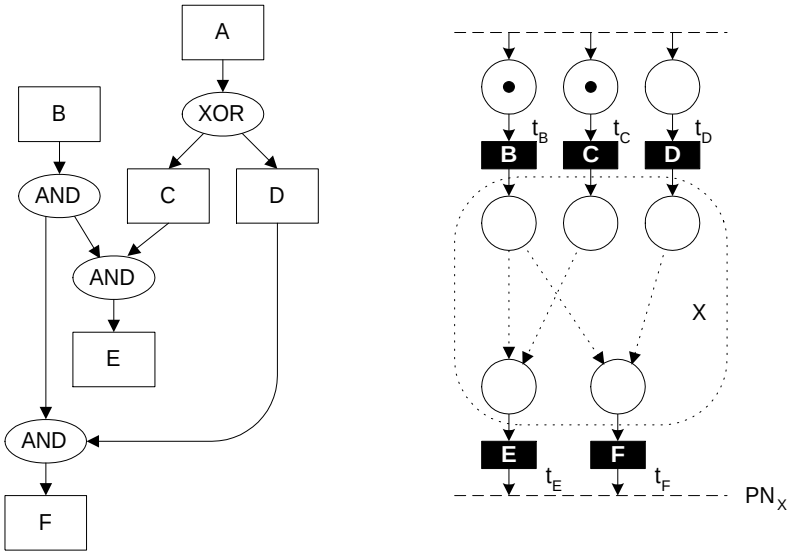

Figure 6.5: Standard Workflow Model with a deadlock

[1] We are not aware of any commercial workflow engine with this capability.

Proof:

Consider the Standard Workflow model $\mathcal{W}$ of Figure 6.5. The semantics of this workflow specification is as follows. After completing activity A a choice is made between activities C and D. At the same time activity B can be performed. If C is chosen and completed along with B, activity E can be performed. If D is chosen and completed along with B, activity F can be performed.

The rest of the proof is analogous to the proof of Theorem 5.4.3. Using the same argumentation we have that in any net PN_x that is bisimulation equivalent to $\mathcal{W}$ there must be transitions labelled B, C, D, E and F (let us call these transitions t_B, t_C, t_D, t_E and t_F). Furthermore if M_1 is a reachable marking of PN_x such that it enables transitions t_B and t_C and no other labelled transitions there must be a firing sequence σ_1 such that $M_1 \stackrel{\sigma_1}{\Longrightarrow} M_2$ and M_2 is a marking that enables transition t_E and no other labelled transition. Similarly if M_3 is a reachable marking of PN_x such that it enables transitions t_B and t_D and no other labelled transitions there must be a firing sequence σ_2 such that $M_3 \stackrel{\sigma_2}{\Longrightarrow} M_4$ and M_4 is a marking that enables transition t_F and no other labelled transition (this is shown in the right diagram of Figure 6.5). The subnet X of this diagram fulfils the conditions of Lemma 5.2.2 (Selective Synchronizer) and we have that PN_x cannot be free-choice or it deadlocks.

□

Theorem 6.2.1 may strike the reader as controversial as deadlock in a specification would always seem to be undesirable. However, the theorem shows that from an expressiveness point of view it is advantageous to be able to instruct a workflow engine what to do in case it encounters a deadlock at runtime. If this option were present in the engine, deadlock could be used as a constructive tool to help design processes that otherwise can not be specified.

6.3 Advanced Synchronization

Standard Workflow models support two types of merge constructs: the AND-Join and the OR-Join. There exist business patterns though that are hard or impossible to capture using these types of merges only. An example of such a pattern is the *discriminator* introduced in Section 3.2.

The discriminator is a merge construct with a fairly straightforward intuitive semantics. It behaves like an OR-Join in the sense that it is nonsynchronizing, an incoming branch can fire the activity following the discriminator, but it is different in the sense that the subsequent activity should not be fired by every incoming branch, only by the one that finishes first.

Consider the simplest process model using the discriminator construct shown in the left diagram of Figure 6.6. In this model, from the initial marking enabling activities A and B the following scenarios are possible:

1. Activity A is completed. Activity C gets enabled and the process finishes when both activities B and C are completed.
2. Activity B is completed. Activity C gets enabled and the process finishes when both activities A and C are completed.
3. Activities A and B are completed before activity C is started. The process finishes once activity C completes. Note that activity C is enabled as soon as either A or B completes.

The important feature of the discriminator in this model is that activity C can be done only once. Formally this behaviour can be captured by the Petri net system PN_D in Figure 6.6 (note that this net system is not free-choice).

The following theorem shows that the discriminator adds expressive power to Standard Workflow Models, as it is inherently non free-choice.

Theorem 6.3.1 (*the discriminator adds expressive power*) There is no Standard Workflow Model equivalent to the Petri net system PN_D of Figure 6.6.

Proof:

Suppose that there is a deadlock-free, free-choice Petri net that is bisimulation equivalent to $\mathcal{W}$. Let us refer to this net as $\mathcal{S}$. This net has to have a transition labelled A and a transition labelled B. We will call these transitions t_A and t_B respectively.

Let M_{AB} be a reachable marking of $\mathcal{S}$ that enables transitions t_A and t_B. Application of the bisimulation game yields that there must be be firing sequence σ_1 such that $M_{AB} \stackrel{t_A\sigma_1}{\longrightarrow} M_{BC}$ and M_{BC} is a marking of $\mathcal{S}$ that enables transition t_B and a transition labelled C (let us call it t_{C1}) but does not enable t_A (or any other transition labelled with A).

Similarly there must be firing sequence σ_2 such that $M_{AB} \stackrel{t_B\sigma_2}{\longrightarrow} M_{AC}$ and M_{AC} is a marking of $\mathcal{S}$ that enables transition t_A and a transition labelled C (let us call it t_{C2}) but does not enable t_B.

Consider now the simulation scenario in which from marking M_{AB} of $\mathcal{S}$, the firing sequence $t_B\sigma_2$ is performed resulting in marking M_{AC} (see net PN_X of Figure 6.6). If it was possible from marking M_{AC} to perform the firing sequence

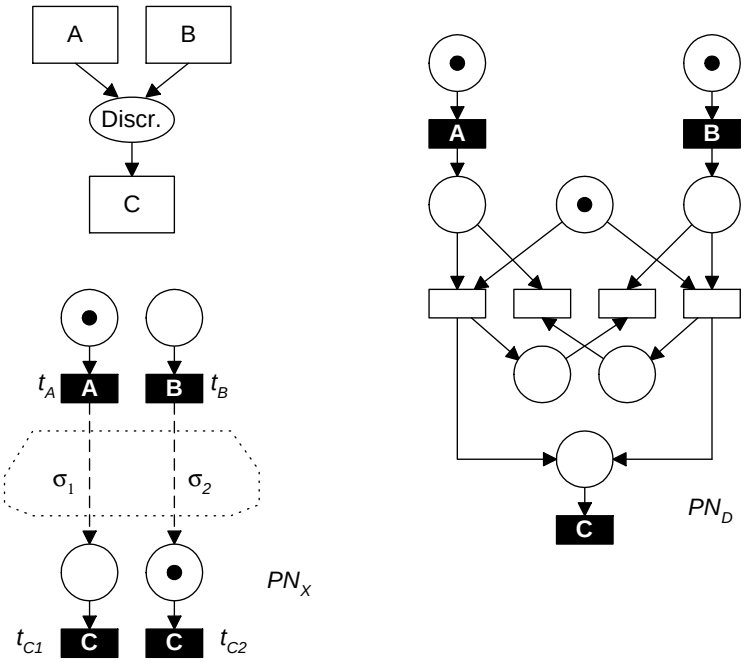

Figure 6.6: Illustration of the discriminator proof

$t_A\sigma_1$, it would be possible to fire both transitions t_{C1} and t_{C2} (if $t_{C1} = t_{C2}$ then it would be possible to fire that transition twice). As that would make $\mathcal{S}$ not equivalent to $\mathcal{W}$, consider the first transition in σ_1 that cannot be fired. If it is possible to enable it by firing some other non-labelled transitions, consider the next transition in σ_1 for which this is impossible. Let us refer to this transition as t_q. This transition must have at least one token in one of its input places. But as $\mathcal{S}$ is free-choice, any other transition that shares its input places with t_q must share all its input places with t_q and therefore it cannot be fired either. As there is no possibility to remove the token from one of the input places of t_q, there is a firing sequence from the marking M_{AC} that results $\mathcal{S}$ to be in deadlock and hence it is not equivalent to $\mathcal{W}$.

□

Considering the semantics of the discriminator in the more general case raises the question as to how it should behave in loops. The simplest solution would be to allow the first incoming branch to trigger the activity following the discriminator and ignore all the other branches from then on. Clearly though this causes a deadlock when the discriminator is used in a loop. A more sophisticated approach would be to allow the first incoming branch to trigger the activity following the discriminator, and to keep track of the other branches. Once all branches have completed, the discriminator is

"reset" and the next incoming branch to finish can again trigger it. This semantics is captured formally by the Petri net shown in Figure 6.7.

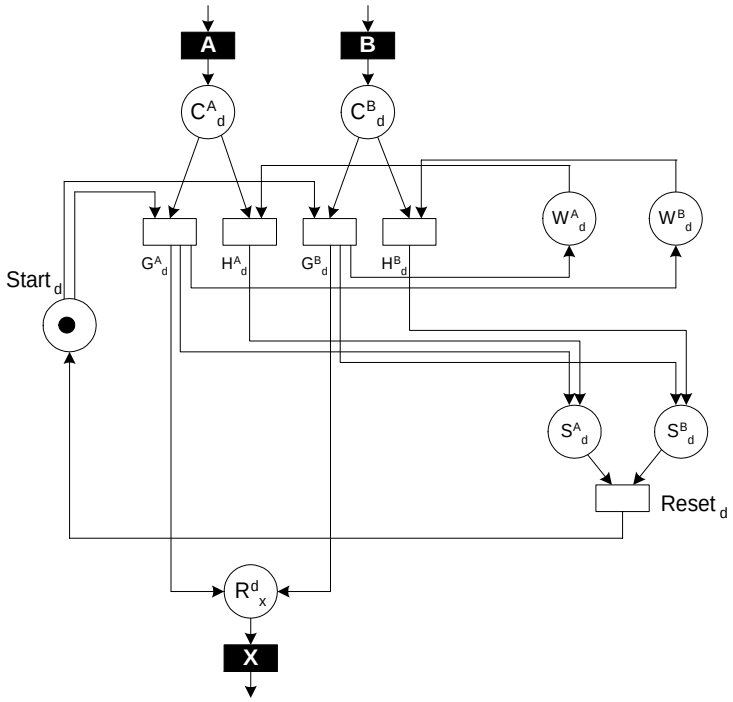

Figure 6.7: Petri net semantics of the discriminator

In Figure 6.7 two activities A and B are shown, which are input to a discriminator d (the schema extends in a natural way to the case of n incoming branches). The place named $Start_d$ initially contains a token (this place is a status place as used in Definition 6.3.3). This represents the situation that the discriminator is waiting for one of its incoming branches to finish. When the first incoming branch finishes, say activity A, activity X is enabled, a token is produced for the place W_d^B to represent the fact that the discriminator still needs to wait for activity B before it can be reset, and a token is placed in place S_d^A so that the fact is remembered that the branch with activity A was already "seen". The completion of B now does not lead to another instance of activity X, rather a token is removed from W_d^B and put in S_d^B. As both branches have now been executed, tokens can be removed from S_d^A and S_d^B and a token can be put in $Start_d$, representing the fact that the discriminator is reset and ready for another iteration. Note that this semantics works well for models that are not guaranteed to be safe, for example the completion of two instances of activity A before an instance of activity B is completed, simply results in the first instance enabling activity X, and the second instance having to wait for an instance of B before it can enable activity X again.

Definition 6.3.1

Syntactically, a Standard Workflow model with discriminators, is a Standard Workflow model $\mathcal{W}$ with a nonempty set $\mathcal{D}$ of discriminators. Each discriminator has an indegree of at least two and an outdegree of one.

□

Definition 6.3.2

Given a Standard Workflow model $\mathcal{W}$ with discriminators from $\mathcal{D}$, the corresponding, marked, labelled, Petri system $PN_{\mathcal{W}} = \langle P'_{\mathcal{W}}, T'_{\mathcal{W}}, F'_{\mathcal{W}}, L'_{\mathcal{W}}, M'_{\mathcal{W}} \rangle$ is defined by:

$$
\begin{aligned}
P'_{\mathcal{W}} &= P_{\mathcal{W}} \cup \\
&\quad \{w_d^x \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup && \#\text{“waiting” places}\# \\
&\quad \{s_d^x \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup && \#\textit{branches already seen}\# \\
&\quad \{Start_d \mid d \in \mathcal{D}\} && \#\text{“start” places}\#
\end{aligned}
$$

$$
\begin{aligned}
T'_{\mathcal{W}} &= T_{\mathcal{W}} \cup \\
&\quad \{G_d^x \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup && \#\textit{transitions to trigger discriminator}\# \\
&\quad \{H_d^x \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup && \#\textit{transitions not to trigger discriminator}\# \\
&\quad \{Reset_d \mid d \in \mathcal{D}\} && \#\text{“reset” transitions}\#
\end{aligned}
$$

$$L'_{\mathcal{W}} = L_{\mathcal{W}}$$

$$
\begin{aligned}
F'_{\mathcal{W}} &= F_{\mathcal{W}} \cup \\
&\quad \{(Start_d, G_d^x) \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup \\
&\quad \{(Reset_d, Start_d) \mid d \in \mathcal{D}\} \cup \\
&\quad \{(c_d^x, G_d^x) \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup \\
&\quad \{(c_d^x, H_d^x) \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup \\
&\quad \{(G_d^x, w_d^y) \mid y \neq x \wedge d \in \mathcal{D} \wedge y \in in(d) \wedge x \in in(d)\} \cup \\
&\quad \{(G_d^x, s_d^x) \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup \\
&\quad \{(G_d^y, r_x^d) \mid d \in \mathcal{D} \wedge y \in in(d) \wedge x \in out(d)\} \cup \\
&\quad \{(w_d^x, H_d^x) \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup \\
&\quad \{(H_d^x, s_d^x) \mid d \in \mathcal{D} \wedge x \in in(d)\} \cup \\
&\quad \{(s_d^x, Reset_d) \mid d \in \mathcal{D} \wedge x \in in(d)\}
\end{aligned}
$$

The initial marking $M'_{\mathcal{W}}$ assigns a single token to each of the places in $\{i_x \mid x \in \mathcal{I}\}$ and to each of the places in $\{Start_d \mid d \in \mathcal{D}\}$. □

Definition 6.3.1 raises an interesting question regarding the termination of a workflow model containing a discriminator. According to definition 4.1.11, any workflow containing a discriminator will never terminate as the token in place Start_d cannot be removed.

When faced with constructs utilizing tokens that keep track of the state of these constructs, rather than the state of the process, the definition of termination needs to be adapted.

Definition 6.3.3 (*relaxed termination for advanced workflows*)
Refer to places that contain tokens in the initial marking of the corresponding Petri net of some workflow specification, but do not correspond to initial places of workflow elements, as status places. *The workflow specification can terminate iff from the initial marking of its corresponding Petri net system a marking can be reached, where only status places contain tokens.* □

It is possible to assign a meaningful semantics to the concept of a discriminator for synchronizing languages. However, there are multiple choices. One could define the discriminator such that it passes on the first token that it receives and ignores tokens from the other activities (till every such activity has generated a token in which case the next cycle could start), or it could be defined in such a way that it passes on the first true-token and waits for tokens from the other activities, but generates a false-token when it receives false-tokens from each of its input activities. Other interpretations are possible as well, but as to the best of our knowledge there is no commercially available workflow system that uses a synchronizing strategy and provides a support for the discriminator, this issue will not be explored further.

6.4 Decomposition

Most process modelling techniques support a form of decomposition. The notion of decomposition is however overloaded and used to achieve, among others, the following:

- The ability to structure the model, making it more effective for communication purposes;
- The ability to specify iterative structures. This is particularly important for synchronizing languages as they cannot directly support loops (see Section 5.4);
- The ability to specify recursion. As shown in [HO99], recursive decomposition adds expressive power as it allows for the specification of some context-free languages (e.g. the language of palindromes) that are not Petri net languages;
- The ability to synchronize multiple instances properly. This is, however, linked to the termination strategy supported and as such is not necessarily something specific for decomposition; This issue will be explored in more depth in this section;

- The ability to thread multiple instances. This issue will also be further explored in the remainder of this section.

Clearly, decomposition is worthy of an in-depth study in itself. Serving so many purposes, it is likely that multiple fundamentally different concepts are at its roots. In the context of this thesis, we will not provide a formal treatment of this notion, but illustrate how it is used to deal with issues related to multiple instances. A formal semantics for decomposition that behaves properly in the context of multiple instances is non trivial. Decomposition as found in various variants of Petri nets, such as Hierarchical Colored Petri nets (see e.g. [Jen98]) or hierarchical nets as presented in [CK81], cannot synchronize properly on multiple instances and do not create multiple threads for each instance of a decomposed activity (hence different threads may interfere). This is not to say that a formal semantics of decomposition cannot be given in e.g. Colored Petri nets, it is just that the characteristics of decomposition are very different and lead to relatively complicated formal semantics.

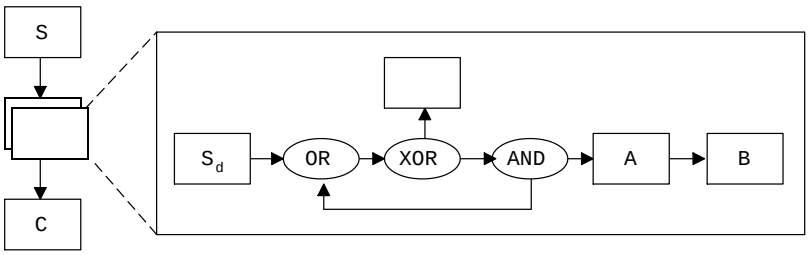

Figure 6.8: A decomposed activity with multiple instances

Let us first focus on the issue of multiple instances. To this end, consider the workflow represented in Figure 6.8. In this workflow, a decomposed activity is followed by an activity named C. The decomposition contains a loop, which, if entered, starts an activity A (followed by an activity B), and in parallel starts the beginning of the loop again. Assuming a lazy termination strategy, the trace set of this workflow is the set of strings consisting of an equal number of a's and b's followed by a single c. Every prefix of any such string has at least as many occurrences of a as of b. Clearly, for every natural number n the string $a^n b^n c$ is an element of the trace set of this workflow. Such a string is called a *counter* and it is well known that there is no finite Petri net that can generate it (see e.g. [Tau89] p. 120; the theorem goes back to [Age75]). This shows, informally, that hierarchical decomposition combined with a lazy termination strategy adds expressive power.

Another aspect of decomposition is the ability to thread multiple instances. To explain this concept consider Figure 6.9. In the Standard Workflow model shown in the lefthand side of this figure, two initial activities are followed by an OR-Join and an

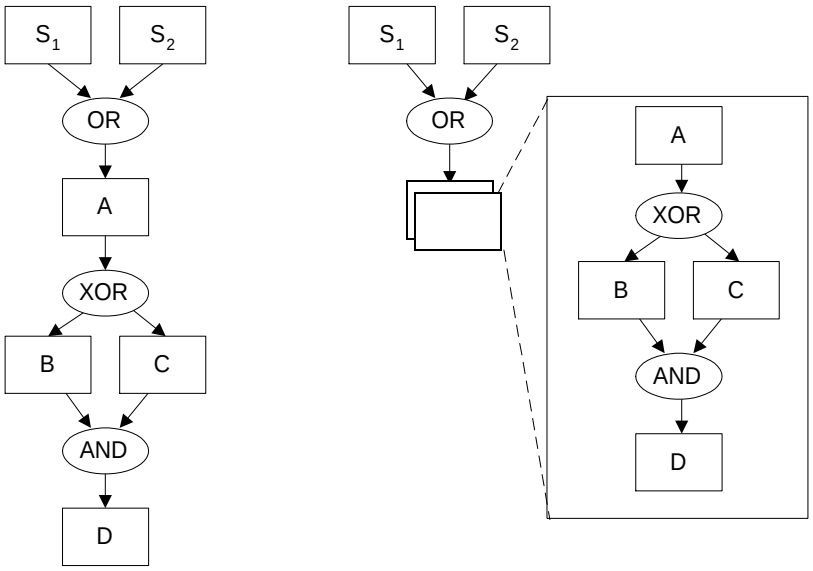

Figure 6.9: Two workflow models which are not equivalent

activity named A. Assuming that this is a Standard Workflow model, during execution this will lead to two instances of activity A being created. After activity A, a choice is made for activities B or C. Activity D then awaits completion of both these activities. The workflow shown in the right-hand side of Figure 6.9 looks similar, except that decomposition is used for the remainder of the workflow following the OR-Join.

The fundamental issue separating both workflows is deadlock. According to the semantics assigned to Standard Workflow models, execution of the workflow on the left-hand side does not always lead to a deadlock: both instances of the XOR-Split may lead to a different outcome, in which case the AND-Join will trigger activity D. Some workflow engines (e.g. Verve Workflow), even though they generate multiple instances of activity A (hence it would seem that they follow the standard workflow execution semantics) will deadlock for all possible workflow instances. The reason for this is that once a new instance of activity A is created, it is executed in its own "thread". Every subsequent activity will also be executed in this thread, which means that in this case two instances of the AND-Join will be created (each of which will deadlock). In other words, the semantics is as in the workflow model shown in the right-hand side of Figure 6.9. In this model every instance has a deadlock as every execution of the decomposed activity occurs in its own space, and communication/synchronization between different instances is not possible. The ability to do this multi-threading is often desirable and it is not clear how it can be supported without the use of special block structures.

6.5 Transformations Using Data Flow

So far throughout this thesis the focus was exclusively on control flow. In this section we would like to hint on the issue of data flow in the context of workflow execution. The following informal discussion is not intended to be exhaustive, it rather tries to underline several important issues related to data flow.

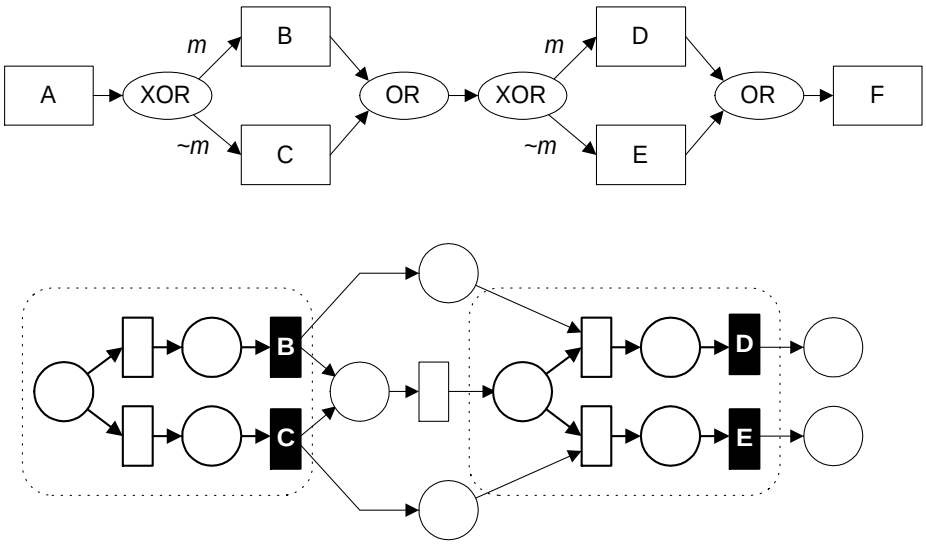

Figure 6.10: Specification using data flow and its Petri net semantics

Consider the simple workflow model depicted in the top diagram of Figure 6.10. There are two choices (XOR-Splits), the first being between performing activities B or C, the second one between performing activities D or E. By using some data predicates these choices are related though. Assuming that only activity A can set the value of m, the traces $abef$ and $acdf$ are not possible. The Petri net representation of this workflow as presented so far in this thesis does not capture the proper behaviour of this workflow model.

The bottom diagram of Figure 6.10 shows a Petri net that better captures the semantics of the workflow. The important feature of this net is the fact that the choice between firing transitions D and E is not free (and consequently, the whole net is not free-choice).

As every Standard Workflow Net is free-choice, the above observation hints on the possibility that usage of data flow to influence control flow of a workflow process can increase the expressiveness. However this comes at a price. Data flow usage decreases workflow processes' suitability. Many workflow engines conceal conditions of XOR-Splits from end-users. It is therefore unclear that there is some extra logic in

a business process diagram and that may often lead to confusion. Furthermore data flow usage makes verification of workflows much harder if not impossible. For tools that perform verification through translation to Petri nets (e.g. Woflan, [AHV97]) it is needed that the translation of a workflow process to a Petri net takes data-related dependencies into account. This is very difficult to do in general case. On the other hand, Flowmake ([SO99]) that performs verification at the workflow process level is unable to take data-related dependencies into account during its workflow model analysis.

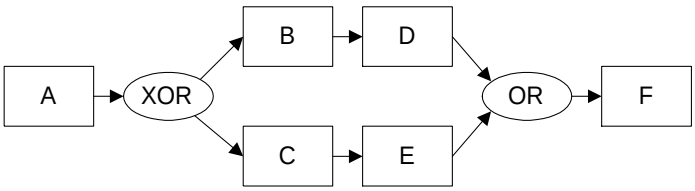

Figure 6.11: Simplified workflow equivalent to workflow of Figure 6.10

For processes where it is possible to find an equivalent representation without concealed data flow dependencies, we strongly advocate that this be done. As an example, Figure 6.11 shows a workflow model equivalent to that of Figure 6.10. There are situations though where this is not possible and this is explored in more details in the remainder of this section.

The common use of data is when transforming of standard workflow models to structured workflow models needs to be performed. In Section 5.3.1 we have presented several transformations useful for this purpose. These transformations rely on node duplication and may result in disproportionally large diagrams. The use of data flow provides workflow modellers with additional transformations. Alternative transformations using data flow for workflows containing entry into a loop structure, exit from a decision structure and entry into a decision structure are shown in Figures 6.12, 6.13 and 6.14 respectively.

As it was shown in Theorem 5.3.2, models containing an exit from a loop structure cannot be transformed to an equivalent structured workflow model using only control flow constructs. The transformation using data flow is depicted in Figure 6.15.

In all these transformations it is important to observe that auxiliary variables (Φ and Θ) are needed to preserve the original values of α and β in case any activities in the model change them.

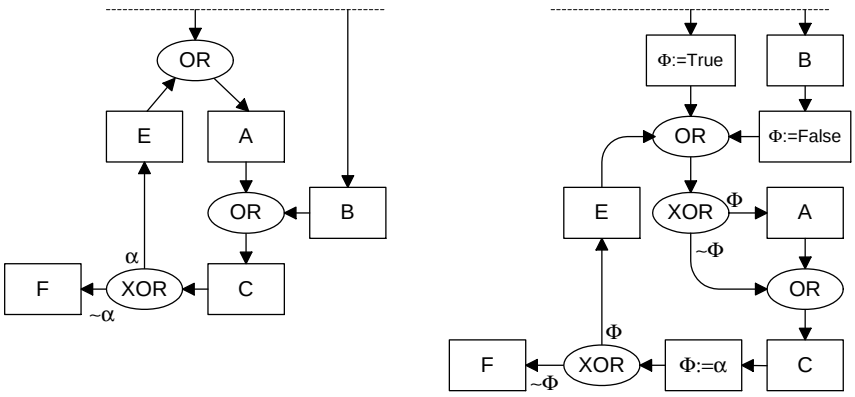

Figure 6.12: Entry into a loop structure

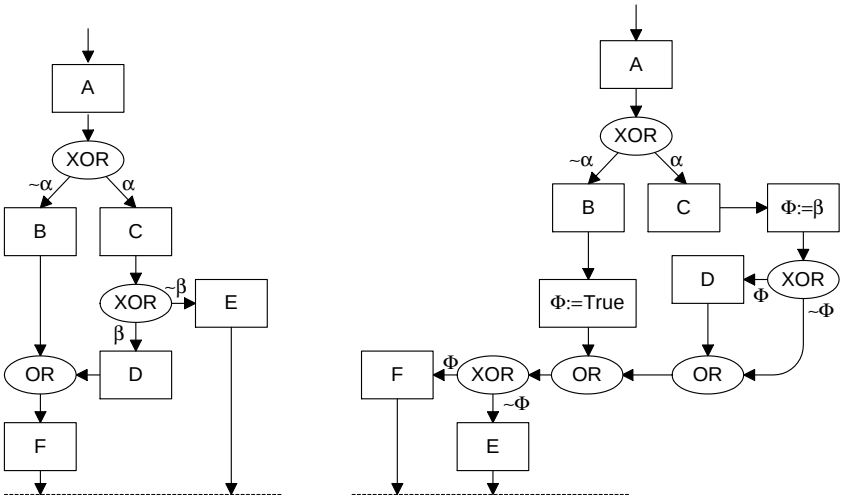

Figure 6.13: Exit from a decision structure

6.6 Summary

In this Chapter we have presented several issues that focus on more advanced aspects of workflow specification beyond the use of standard control flow constructs such as AND-Joins, AND-Splits, OR-Joins and OR-Splits. In Section 6.1 we argue against a *strict* termination policy which is commonly deployed by workflow vendors. Additionally we present a transformation that can be used to transform well-behaved models that have more than one final task into models with one final task. This transformation is useful to any workflow modeller working with a language that requires a unique final task in a workflow process. In Section 6.2 we present theoretical arguments for

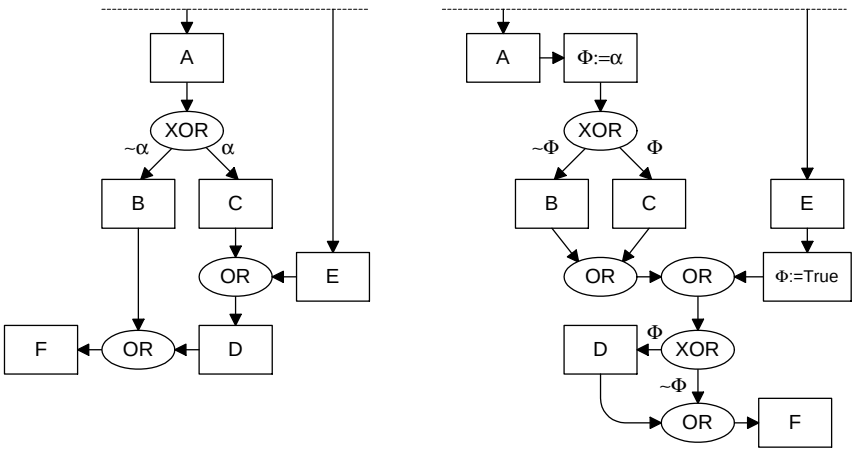

Figure 6.14: Entry into a decision structure

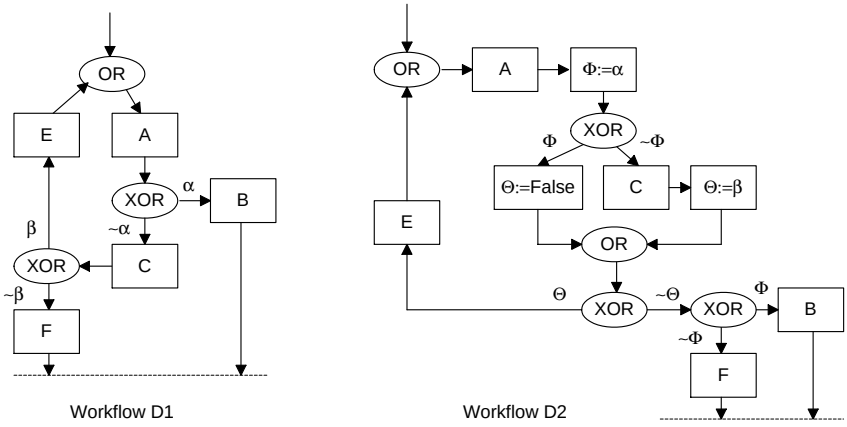

Figure 6.15: Exit from a loop structure

adopting an *active deadlock resolution* strategy. We are not aware of any workflow language that employs such a strategy. In Section 6.3 we provide a proof that a *Discriminator* construct is not possible to implement using Standard Workflow Models. We consider it to be a good argument for adoption of the discriminator construct in a modern workflow modelling language. In Section 6.4 decomposition is studied in more depth and finally in Section 6.5 we provide an insight into using data flow to augment control flow specifications of workflow models. Additionally we present several transformations that can be useful when transforming Standard Workflow Models to Structured Workflow Models.

Chapter 7

Conclusions

Workflow technology, with the advancement of electronic commerce, business-to-customer and business-to-business interactions and an increasing pressure to streamline and automate processes of any enterprise that wants to stay competitive, has become more ubiquitous than ever.

As of the beginning of 2002, the Workflow Management Coalition, arguably the most prominent standardisation body for workflow technology, lists over 280 members, many of them being workflow engine providers. The Workflow and Reengineering International Association (WARIA)[1], a non-profit organization that closely monitors the workflow market, lists over 70 workflow vendors, and undoubtly there are many workflow products that escape its classification.

It is not surprising then, given the lack of a well-established formal foundation to workflow modelling, that many different approaches towards workflow modelling have been proposed and implemented in commercial products. In this Thesis we present eight different commercial products. As we have demonstrated in Chapter 2 through a simple test harness consisting of a set of workflow processes that were then run in every evaluated product, there are substantial differences in interpretation of even the most basic concepts that are used to define the control flow part of a workflow specification.

It is therefore surprising that in discussions on workflow products, emphasis is hardly ever on the workflow languages used, rather focus is almost exclusively on operational and architectural aspects. The standardization efforts of the Workflow Management Coalition with respect to process modelling have fallen short in achieving its stated goal of providing a common process interchange format. In [Wor99a] the WfMC presents WPDL, Workflow Process Definition Language, that, once generated by pro-

[1] http://www.waria.com

cess design tools, is "capable of interpretation in different workflow run-time products". As we have shown through the test harness, the interpretation of seemingly well-understood concepts, varies greatly from product to product.

We feel that one of the fundamental reasons for the existence of many incompatible modelling techniques is a lack of deep understanding of the implications of various approaches to specifying control flow in workflows. In this Thesis we have embarked on the task of providing a milestone for such knowledge and the approach we have taken focuses on three principles of a good modelling technique: suitability, expressiveness and formality.

In Chapter 3 we have indicated requirements for workflow languages through the use of "workflow patterns". To our knowledge no other attempts have been made to collect a structured set of workflow patterns and, as we have demonstrated in that chapter, no workflow product that we have evaluated, supports all identified patterns.

It is difficult to gather quantitative data about the occurrence frequency of each pattern, however given that we have provided real-life examples for each of these patterns, it is likely that some business processes may be hard to automate using the current crop of workflow technology. Patterns involving multiple instances (presented in Section 3.4) and state-based patterns (presented in Section 3.5) seem to present the biggest challenge for workflow products. It is our consulting experience that patterns belonging to the first group (patterns involving multiple instances) are frequently encountered in real-life.

The other goal that we have tried to achieve with our patterns work was to suggest different implementation strategies for each of the patterns. These solutions have various degrees of suitability and in some cases no solution can be presented hinting at the lack of expressive power in a given modelling language.

In Chapter 4, in order to formally investigate the expressive power of different modelling approaches, we have presented a Petri net based formal foundation along with an equivalence notion that can be used to compare different process models. Then in Chapter 5 we have evaluated the four most common evaluation strategies with respect to their expressive power. The following strategies were identified.

1. *Standard.* This strategy is used by Verve and Forté Conductor. It allows for creation of concurrent multiple instances of an activity using a standard OR-Join construct and it allows for specification of arbitrary cycles. The main downside is that models employing this strategy may deadlock.

2. *Safe.* This strategy is similar to Standard, the main difference being lack of support for concurrent multiple instances of an activity. This strategy is used by HP Changengine, Staffware and Fujitsu i-Flow. Again, models employing this strategy may deadlock.

3. *Structured.* This strategy imposes syntactical restrictions on a workflow model. It is not possible to model arbitrary cycles and the use of standard control flow modelling constructs such as joins and splits is restricted. The advantage of this approach is that models employing this strategy never deadlock, nor do they result in creation of concurrent multiple instances. The fundamental disadvantage is a decreased expressive power when compared to other strategies.

4. *Synchronizing.* This strategy takes a fundamentally different view to workflow execution and from a theoretical point of view its expressive power is not comparable to the Standard Strategy. This approach is used by MQSeries Workflow.

Finally in Chapter 6 we have investigated some other issues associated with choices that workflow engine designers are likely to face. More specifically we have shown that it is possible to convert any workflow model that does not deadlock and does not create concurrent multiple instances into an equivalent workflow model that has only one final task. The resulting model, however, is more complex and cluttered, hence less suitable. Then we have shown that from an expressiveness point of view it is desirable for a workflow engine to dynamically detect deadlocks. None of the workflow engines evaluated in the course of this work has this facility. Then we have demonstrated that some more advanced control construct, notably the discriminator, increases the expressive power of a workflow language. Finally we have suggested possible issues related to the use of decomposition in a workflow specification and provided a discussion on the use of data flow to enhance control flow specification. It is our contention that models that do not use data flow to affect control flow specification are more suitable and are easier understood, however, we concede that faced with limitations in expressive power, sometimes it is necessary to use data flow to model certain processes.

It is our hope that the results presented in this thesis will aid both workflow analysts and workflow engine designers as well as provide a "bridge" between academia and industry. For workflow analysts the results, among others, will allow them to understand the inherent limitations of the languages they need to specify their workflows in. For workflow engine designers the results suggest directions for improving the expressive power of their engine. For researchers this work may provide an insight into formal foundations of workflow specifications.

7.1 Towards a Better Design of a Workflow Language

As indicated in the introduction to this thesis, it was never our intention to provide a specification for a new workflow specification language. Nevertheless the results

presented in the thesis provide valuable suggestions for designers of new workflow engines. With a view of achieving greater expressiveness and suitability the following is a list of suggestions for the design of the control flow perspective of a new workflow specification language.

1. A workflow language should use an explicit notation for control flow constructs. Languages allowing implicit notation (for example an activity with more than one outgoing transition implements an AND-Split while an activity with more than one incoming transition implements an OR-Split) may cause interpretation problems for end-users thus reducing their suitability.

2. The workflow engine should employ the Standard Workflow Model evaluation strategy which is formally given in Section 4.1.1. As discussed in Chapter 5 this evaluation strategy has a distinctive advantage over alternative strategies in terms of the expressiveness of the model. It is more expressive than the Safe and Structured evaluation strategies and even though it is not comparable to the Synchronizing strategy, we feel that it is much less restrictive allowing for the usage of arbitrary loops in a workflow model as well as providing a natural approach to modelling multiple instances.

3. An XOR-Split construct should be used instead of an OR-Split. Given the choice of the Standard evaluation strategy, the OR-Split construct does not have a corresponding join construct that will not deadlock or not result in unwanted multiple instances. In practice it is possible to use the data predicates to ensure that an OR-Split has the semantics of an XOR-Split or an AND-Split, however we believe that providing these two constructs explicitly adds to the suitability of the language and enhances the potential for automated verification.

4. The workflow engine should exploit the full expressive power of the Standard evaluation strategy and not impose any unnecessary syntactical restrictions, e.g. restrictions on cycles in a workflow graph or rules dictating allowable combinations of splits and joins. As shown in Sections 5.3 and 6.5 transformation of arbitrary models to structured models may not always be possible and if they can be done, they usually affect the model's suitability.

5. The workflow engine should have a relaxed termination policy (see Section 6.1), in other words the workflow specification language should not mandate the use of special "terminal" tasks. Rather, the execution of a workflow model should be considered complete when there are no remaining active threads. This allows for simpler models as well as a natural way of achieving synchronisation of multiple instances (see Pattern 15 in Section 3.4).

6. The workflow engine should support active deadlock detection (see Section 6.2) and allow workflow designers to specify in advance what the workflow engine should do in case of runtime deadlock. In particular, this allows treating deadlock as successful termination and it was shown that such a feature increases the expressive power of the language.

7. The workflow engine should support a notion of decomposition that allows for thread separation and employs a relaxed termination strategy in order to synchronise multiple instances (see Section 6.4). We feel that decomposition should not be the only mechanism through which iterative structures can be specified.

8. The multiple instances of a single activity invoked as a result of, for example, an OR-Join preceded by an AND-Split construct should run in the same thread (as noted above decomposition can be used to provide thread separation, if needed). This allows, for example, in a process where two instances of an activity A need to be synchronized with two instances of an activity B to synchronize any instance of A with any instance of B (see Figure 5.15 in Section 5.2).

9. The workflow engine should support advanced synchronisation constructs such as the discriminator. We concede that while adding more advanced constructs may increase suitability and expressive power, the verification becomes more problematic and care should be taken to balance expressiveness and simplicity of the language. To this end the inclusion of the basic discriminator construct (which enhances the expressive power of the Standard evaluation strategy) and omission of the more general n-out-of-m merge (which does not add an extra expressive power over the discriminator) may be a viable approach.

10. The workflow engine should support the deferred choice construct (see Section 3.5). As indicated in the Introduction, some authors propose the use of Petri nets for workflow specification. In particular, in [Aal98a] van der Aalst provides three reasons for such an approach, namely the fact that Petri nets are formal, that they have associated analysis techniques, and they support an explicit notion of state. Our approach has been to use Petri nets as a formal foundation for workflows rather than as a workflow specification language in itself. This increases suitability while maintaining the benefits of having a formal, rigorous semantics and being able to leverage from existing results of Petri net theory. Our approach does not provide an explicit notion of state, however, as we have demonstrated in Section 3.5, the inclusion of the deferred choice as part of the specification language allows for the successful implementation of many state-based patterns.

7.2 Future Work

In this Thesis focus was on the control flow perspective only, apart from short discussion in Section 6.5. We believe that it is important that control flow and data flow are separated out as much as possible, as workflows become harder to (formally) analyse and understand, the moment part of their control flow is "hidden" in the data flow. Hence, it is imperative to first understand expressiveness issues within the control flow perspective before considering data flow. Nevertheless, the inclusion of data flow and its implications for expressiveness are considered an important avenue for further research. Other perspectives, in particular the resource perspective dealing with organizational aspects of workflow specifications are also worth considering.

For the theoretical framework presented in this Thesis we have made several assumptions that may be worth re-considering in the future. For example our choice of an equivalence notion can be argued as too relaxed/restrictive and it will be interesting to know how the choice of equivalence affects the results presented in Chapters 5 and 6. We have also assumed that the execution of an activity is atomic - in real life workflows it may be possible that some activities are long-running, possibly for many months, whilst some are short, lasting no more than few milliseconds. This aspect of workflow execution introduces another layer of complexity for control flow specification. For long-running activities it may be imperative to capture their execution state in a more detailed manner hence opening up a new avenue for more advanced control flow dependencies and execution patterns.

The patterns work, by definition, is never complete. As part of this research we have started a community website[2] where we have published all the patterns presented in this thesis as well as invited all people interested in the topic to contribute their own patterns. The response has been very encouraging and we are trying to make every effort to keep the pattern collection up-to-date with suggestions from workflow users. Another interesting avenue for future work related to workflow patterns is to gather quantitative data related to the popularity of each of the patterns thus prioritising the deficiencies in the current approaches towards modelling of workflows. Such work has already begun by Wil van der Aalst through the evaluation of five workflow projects conducted by ATOS/Origin (Utrecht, The Netherlands).

[2] http://tmitwww.tm.tue.nl/research/patterns/

Appendix A

Product Evaluation

Pattern	Score	Motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported using "Static Split" construct.
3 (synch)	+	Supported using "Rendezvous" construct.
4 (ex-ch)	+	Supported using "Branch" construct.
5 (simple-m)	+	Supported using "Branch" construct.
6 (m-choice)	+/-	Indirectly supported through the combination of other constructs.
7 (sync-m)	-	Not supported.
8 (multi-m)	-	Not supported.
9 (disc)	-	Not supported.
10 (arb-c)	-	Very limited support using "Goto" construct.
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)	-	Supported through the "Terminate" construct.

Table A.1: Visual WorkFlo

Pattern	Score	Motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported through an activity with many outgoing transitions having no associated conditions.
3 (synch)	+	Supported through a "Synchronizer" construct.
4 (ex-ch)	+/-	Supported through an activity with many outgoing transitions having associated conditions such that only one evaluates to "true" during runtime.
5 (simple-m)	+	Supported through an activity with more than one incoming transition.
6 (m-choice)	+	Supported through an activity with more than one outgoing transition.
7 (sync-m)	-	Not supported.
8 (multi-m)	+	Supported through an activity with more than one incoming transition.
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)	+	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 a final activity.

Table A.2: Verve Workflow

Pattern	Score	Motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported through an activity with multiple outgoing transitions.
3 (synch)	+	Supported through a Wait Step.
4 (ex-ch)	+	Supported through a Decision Step.
5 (simple-m)	+	Supported through an activity with multiple incoming transitions.
6 (m-choice)	+/-	Supported through a combination of AND-Splits and XOR-Splits.
7 (sync-m)	-	Not supported.
8 (multi-m)	-	Not supported.
9 (disc)	-	Not supported.
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)	-	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. However it is possible to use the "withdraw" facility to achieve approximate solution.
17 (int-par)	-	Not supported.
18 (milest)	-	Not supported.
19 (can-a)	+	Supported through a "withdraw" facility.
20 (can-c)	-	Not supported.

Table A.3: Staffware

Pattern	Score	Motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported through an activity with many outgoing transitions having no associated conditions.
3 (synch)	+	Supported through an activity with many incoming transitions.
4 (ex-ch)	+/-	Supported through an activity with many outgoing transitions having associated conditions such that only one evaluates to "true" during runtime.
5 (simple-m)	+	Supported through an activity with many incoming transitions.
6 (m-choice)	+	Supported through an activity with more than one outgoing transition.
7 (sync-m)	+	Supported through an activity with many incoming transitions.
8 (multi-m)	-	Not supported.
9 (disc)	-	Not supported.
10 (arb-c)	-	Not supported.
11 (impl-t)	+	Directly supported. The workflow instance terminates if all of the corresponding branches have terminated.
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.

Table A.4: MQSeries Workflow

Pattern	Score	Motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported through an activity with many outgoing transitions having no associated conditions.
3 (synch)	+	Supported through an activity with many incoming transitions. This activity needs to have a trigger condition defined so that it synchronises incoming threads.
4 (ex-ch)	+	Supported through an activity with many outgoing transitions and a switch that allows only one of the transitions to fire.
5 (simple-m)	+	Supported through an activity with many incoming transitions. This activity needs to have a special trigger condition defined so that it merges incoming threads.
6 (m-choice)	+	Supported through an activity with many outgoing transitions and a switch that allows multiple transitions to fire.
7 (sync-m)	-	Not supported.
8 (multi-m)	+	Supported through an activity with many incoming transitions. This activity needs to have a trigger condition defined so that it merges incoming threads.
9 (disc)	+/-	Supported through an activity with many incoming transitions. This activity needs to have a custom trigger defined.
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)	-	Not supported.
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.

Table A.5: Forté Conductor

Pattern	Score	Motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported through a Route Node with more than one outgoing transition.
3 (synch)	+	Supported through a Route Node with more than one incoming transition and a rule such that synchronization is achieved.
4 (ex-ch)	+	Supported through a Route Node with more than one outgoing transition.
5 (simple-m)	+	Supported through a Route Node with more than one incoming transition and a rule such that merging is achieved.
6 (m-choice)	+	Supported through a Route Node with more than one outgoing transition.
7 (sync-m)	-	Not supported.
8 (multi-m)	-	Not supported.
9 (disc)	+	Supported through a Route Node with more than one incoming transition and a rule such that merging is achieved.
10 (arb-c)	+	Directly 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)	-	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 an Abort Node.

Table A.6: HP Changengine

Pattern	Score	Motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported through an OR Node with many outgoing transitions.
3 (synch)	+	Supported through an AND node.
4 (ex-ch)	+	Supported through a Conditional Node.
5 (simple-m)	+	Supported through an activity with many incoming transitions.
6 (m-choice)	+/-	Supported through a combination of OR Node and Conditional Node.
7 (sync-m)	-	Not supported.
8 (multi-m)	-	Not supported (no concurrent multiple instances are permitted).
9 (disc)	-	Not supported.
10 (arb-c)	+	Directly supported.
11 (impl-t)	-	Not supported.
12 (mi-no-s)	+	Supported through a 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.

Table A.7: Fujitsu i-Flow

Pattern	Score	Motivation
1 (seq)	+	Directly supported.
2 (par-spl)	+	Supported through a "fork" construct.
3 (synch)	+	Supported through a "join" construct.
4 (ex-ch)	+	Supported through a "condition" construct.
5 (simple-m)	+	Supported through a "condition" construct.
6 (m-choice)	+/-	Supported through a combination of "fork" and "condition" .
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 behaviour 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.

Table A.8: SAP R/3 Workflow

Appendix B

Petri Nets: Notations and Definitions

This section introduces basic Petri net terminology and notations and is adapted from [Pet81] and [DE95]. Readers familiar with Petri nets can skip this section.

The classical Petri net is a directed bipartite graph with two node types called *places* (graphically represented by circles) and *transitions* (graphically represented by thick lines). The nodes are connected via directed *arcs*.

A *net* is a tuple $PN = \langle P, T, F \rangle$ where P and T are finite disjoint sets of *places* and *transitions* respectively, and $F \subset (P \times T) \cup (T \times P)$ is a set of *arcs* (flow relation).

A place p is called an *input place* of a transition t iff there exists a directed arc from p to t. Place p is called an *output place* of transition t iff there exists a directed arc from t to p. We use $\bullet t$ to denote the set of input places for a transition t. The notations $t\bullet$, $\bullet p$ and $p\bullet$ have similar meanings, e.g. $p\bullet$ is the set of transitions sharing p as an input place.

At any time a place contains zero or more *tokens*, drawn as black dots. The *state* M, often referred to as marking, is the distribution of tokens over places, i.e. $M \in P \rightarrow \mathbb{N}$. We will represent a state as follows: $1p_1 + 2p_2 + 1p_3 + 0p_4$ is the state with one token in place p_1, two tokens in p_2, one token in p_3 and no tokens in p_4. We can also represent this state as follows: $p_1 + 2p_2 + p_3$. To compare states, we define a partial ordering. For any two states M_1 and M_2, $M_1 \leq M_2$ iff for all $p \in P$: $M_1(p) \leq M_2(p)$

The number of tokens may change during the execution of the net. Transitions are the active components in a net: they change the state of the net according to the following *firing rule*:

(1) A transition t is said to be *enabled* iff each input place p of t contains at least one token.

(2) An enabled transition may *fire*. If transition t fires, then t *consumes* a token from each input place p of t and *produces* a token for each output place p of t.

A *Petri net system* (or net system) is a tuple $N = (PN, M_0)$, where PN is a Petri net and M_0 is an *initial* marking.

A *labeled* Petri net is a tuple $\langle P, T, F, L\rangle$ where (P, T, F) is a Petri net and L is a mapping that associates to each transition t a label $L(t)$ taken from some given set of actions $\mathcal{A}$.

Given a labeled Petri net $PN = (P, T, F, L)$ and a state M_1, we have the following notations:

- $M_1 \xrightarrow{t}_{PN} M_2$: transition t is enabled in state M_1 and firing t in M_1 results in state M_2
- $M_1 \longrightarrow_{PN} M_2$: there is a transition t such that $M_1 \xrightarrow{t}_{PN} M_2$
- $M_1 \xrightarrow{PN}_{PN} M_n$: the firing sequence $PN = t_1 t_2 t_3 \ldots t_{n-1} \in T^*$ leads from state M_1 to state M_n, i.e. $M_1 \xrightarrow{t_1}_{PN} M_2 \xrightarrow{t_2}_{PN} \ldots \xrightarrow{t_{n-1}}_{PN} M_n$
- $M_1 \xrightarrow{a}_{PN} M_2$: there is a transition t labeled a, i.e. $L(t) = a$ that is enabled in state M_1 and firing t in M_1 results in state M_2

A state M_n is called *reachable* from M_1 (notation $M_1 \xrightarrow{*}_{PN} M_2$) iff there is a firing sequence $PN = t_1 t_2 \ldots t_{n-1}$ such that $M_1 \xrightarrow{PN}_{PN} M_n$. The subscript $_{PN}$ is omitted if it is clear which Petri net is considered. Note that the empty firing sequence is also allowed, i.e. $M_1 \xrightarrow{*}_{PN} M_1$.

We use (PN, M_0) to denote a Petri net system PN with an initial state M_0. A state M is a *reachable state* of (PN, M_0) iff $M_0 \xrightarrow{*} M$.

A marking M_h is a *home marking* iff for every reachable marking M, $M \xrightarrow{*} M_h$.

A marking M is an *empty marking* iff for every place s we have $M(s) = 0$. We will use the symbol $M^{\varnothing}$ for the empty marking.

A Petri net is *safe* if $M(s) \leq 1$ for every place s and every reachable marking M.

A Petri net is *bounded* if the set of reachable markings is finite.

A Petri net is *free-choice* iff, for all transitions $t_1, t_2 \in T$, either $\bullet t_1 \cap \bullet t_2 = \varnothing$ or $\bullet t_1 = \bullet t_2$.

Bibliography

[AAA+95] G. Alonso, Divyakant Agrawal, Amr El Abbadi, C. Mohan, Roger Gunthor, and Mohan Kamath. Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Managemen. In *Proceedings of the IFIP WG8.1 Working Conference on Information Systems Development for Decentralized Organizations. Trondheim, Norway, August 1995.*, 1995.

[AAH98] N. R. Adam, V. Atluri, and W. Huang. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems (JIIS), Special Issue on Workflow and Process Management*, 10(2):131–158, March/April 1998.

[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. Formalization and verification of event-driven process chains. Technical Report Computing Science Reports 98/01, Eindhoven University of Technology, Eindhoven, the Netherlands, 1998.

[Aal98c] 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.

[ABHK00] W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, and B. Kiepuszewski. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *Fifth IFCIS International Conference on Cooperative Information Systems (CoopIS'2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29, Eilat, Israel, September 2000. Springer-Verlag.

[Age75] T.K.M. Agerwala. *Towards a theory for the analysis and synthesis of systems exhibiting concurrency.* PhD thesis, The John Hopkins University, Baltimore, Maryland, 1975.

[AHH94] W.M.P. van der Aalst, K.M. van Hee, and G.J. Houben. Modelling and Analysing Workflow using a Petri-net based approach. In G. De Michelis and C. Ellis, editors, *Proceedings of Second Workshop on Computer-supported Cooperative Work, Petri nets related formalisms,*, pages 31–50, 1994.

[AHKB00] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Technical Report WP 47, BETA Research Institute, Eindhoven University of Technology, Eindhoven, The Netherlands, August 2000. Submitted for publication in Distributed and Parallel Databases.

[AHV97] W.M.P. van der Aalst, D. Hauschildt, and H.M.W. Verbeek. A Petri-net-based Tool to Analyze Workflows. In B. Farwer, D. Moldt, and M.O. Stehr, editors, *Proceedings of Petri Nets in System Engineering (PNSE'97)*, pages 78–90, Hamburg, Germany, September 1997. University of Hamburg (FBI-HH-B-205/97).

[AHV02] W.M.P. van der Aalst, A. Hirnschall, and H.M.W. Verbeek. An Alternative Way to Analyze Workflow Graphs. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*, Berlin, Germany, 2002.

[AMP94] A. Agostini, G. De Michelis, and K. Petruni. Keeping Workflow Models as simple as possible. In *Proceedings of the workshop on CSCW, Petri-Nets and related formalism. 5th International Conference on Allpication and Theory of Petri-nets*, pages 11–29, Zaragoza, Spain, June 1994.

[AW91] D.E. Avison and A.T. Wood-Harper. Information Systems Development Research: An Exploration of Ideas in Practice. *The Computer Journal*, 34(2):98–112, April 1991.

[BK84] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60:109–137, 1984.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, United Kingdom, 1990.

[BW95] P. Bartheless and J. Wainer. Workfow Systems: a few definitions, a few suggestions. In *Proceedings of the Conference on Organizational Computing Systems*, Milpitas, CA. USA, November 1995.

[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, Gold Coast, Australia, December 1995. Springer-Verlag.

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

[CK81] L.A. Cherkasova and V.E. Kotov. Structured Nets. In J. Gruska and M. Chytil, editors, *Proceedings 10th Symposium on Mathematical Foundations of Computer Science*, volume 118 of *Lecture Notes in Computer Science*, pages 242–251, Strbské Pleso, Czechoslovakia, 1981. Springer.

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

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

[Ell79] C.A. Ellis. Information Control Nets: A Mathematical Model of Office Information Flow. In *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 225–240, Boulder, Colorado, 1979. ACM Press.

[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, Berlin, Germany, 1993. Springer-Verlag.

[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.

[Gen87] H. Genrich. Predicate/Transition Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986 Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, Berlin, Germany, 1987.

[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 Modelling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995.

[Gla90] R.J. van Glabbeek. The linear time-branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of CONCUR'90. Theories of Concurrency: Unification and Extension*, pages 278–297, Berlin, Germany, 1990. Springer-Verlag.

[Gla93] R.J. van Glabbeek. The linear time – branching time spectrum II; the semantics of sequential systems with silent moves (extended abstract). In E. Best, editor, Proceedings *CONCUR'93,* 4^{th} International Conference on Concurrency Theory, Hildesheim, Germany, August 1993, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.

[Gla94] R.J. van Glabbeek. What is branching time semantics and why to use it? In M. Nielsen, editor, *The Concurrency Column*, pages 190–198. *Bulletin of the EATCS* 53, 1994. Also available as Report STAN-CS-93-1486, Stanford University, 1993, and by ftp at `ftp://Boole.stanford.edu/-pub/branching.ps.gz`.

[Gri82] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base.* Publ. nr. ISO/TC97/SC5/WG3-N695, ANSI, 11 West 42nd Street, New York, NY 10036, 1982.

[GV87] R.J. van Glabbeek and F.W. Vaandrager. Petri net models for algebraic theories of concurrency (extended abstract). In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *Proceedings PARLE, Parallel Architectures and Languages Europe, Vol. II: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 224–242, Eindhoven, The Netherlands, June 1987. Springer Verlag.

[HHSW96] Y. Han, J Himminghofer, T. Schaaf, and D. Wikarski. Management of Workflow Resources to Support Runtime Adaptibility and System Evaluation. In *Proceedings of PAKM96 Internaltional Conference on Knowledge Engineering*, Basel, Switzerland, Oct 1996.

[HK99] A.H.M. ter Hofstede and B. Kiepuszewski. Formal Analysis of Deadlock Behaviour in Workflows. Technical Report FIT-TR-1999-03, Queensland University of Technology/Mincom, Brisbane, Australia, April 1999.

[HN93] A.H.M. ter Hofstede and E.R. Nieuwland. Task structure semantics through process algebra. *Software Engineering Journal*, 8(1):14–20, January 1993.

[HO99] A.H.M. ter Hofstede and M.E. Orlowska. On the Complexity of Some Verification Problems in Process Control Specifications. *Computer Journal*, 42(5):349–359, 1999.

[Hof93] A.H.M. ter Hofstede. *Information Modelling in Data Intensive Domains*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[HOR98] A.H.M. ter Hofstede, M.E. Orlowska, and J. Rajapakse. Verification Problems in Conceptual Workflow Specifications. *Data & Knowledge Engineering*, 24(3):239–256, January 1998.

[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.

[Jab94] S. Jablonski. Mobile: A modular workflow model and architecture. In *Proceedings of 4th International Conference on Dynamic Modeling and Information Systems*, Netherlands, 1994.

[Jan94] P. Jančar. Decidability Questions for Bismilarity of Petri Nets and Some Related Problems. In P. Enjalbert, E.W. Mayr, and K.W. Wagner, editors, *STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science*, volume 775 of *Lecture Notes in Computer Science*, pages 581–592, Caen, France, February 1994. Springer-Verlag.

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

[Jen87] K. Jensen. Coloured Petri Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986 Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299, Berlin, Germany, 1987. Springer-Verlag.

[Jen98] Kurt Jensen. An introduction to the practical use of coloured petri nets. In G. Rozenberg and W. Reisig, editors, *Lectures on Petri Nets II: Applications*, volume 1492 of *Lecture Notes in Computer Science*. Springer, 1998.

[KG99] M. Kradolfer and A. Geppert. Dynamic Workflow Schema Evolution based on Workflow Type Versioning and Workflow Migration. In *Proccedings of the Fourth IFCIS International Conference on Cooperative Information Systems (CoopIS99)*, Edinburgh, Scotland, Sep 1999.

[KHA01] B. Kiepuszewski, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Fundamentals of Control Flow in Workflows. Technical Report FIT-TR-2001-01, Queensland University of Technology/Mincom, Brisbane, Australia, January 2001. Conditinally accepted for publication in Acta Informatica.

[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 12th International Conference on Advanced Information Systems Engineering (CAiSE'00)*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445, Stockholm, Sweden, June 2000. Springer.

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

[KT98] G. Keller and Th. Teufel. *SAP R/3 Process Oriented Implementation*. Addison-Wesley Longman, Harlow, United Kingdom, 1998.

[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.

[Lon98] J. Lonchamp. Process Model Patterns for Collaborative Work. In *Proceedings of the 15th IFIP World Computer Congress, Telecooperation Conference, Telecoop'98*, Vienna, Austria, July 1998.

[Low01] D. Lowe et al. *BizTalk Server: The Complete Reference.* McGraw-Hill Professional Publishing, 2001.

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

[LZLC02] H. Lin, Z. Zhao, H. Li, and Z. Chen. A Novel Graph Reduction Algorithm to Identify Structural Conflicts. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, Big Island, Hawaii, January 2002.

[MAGK95] C. Mohan, Gustavo Alonso, Roger Gunthor, and Mohan Kamath. Exotica: A Research Perspective on Workflow Management Systems. *Data Engineering Bulletin*, 18(1):19–26, 1995.

[Mak96] P. Makey, editor. *Workflow: Integrating the Enterprise.* Report of the Butler Group, June 1996.

[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.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, Cambridge, United Kingdom, 1999.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77:541–580, 1989.

[Oul82] G. Oulsnam. Unravelling Unstructured Programs. *Computer Journal*, 25(3):379–387, 1982.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

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

[PRS92] Lucia Pomello, Grzegorz Rozenberg, and Carla Simone. A survey of equivalence notions of net based systems. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 609 of *Lecture Notes in Computer Science*, pages 410–472. Springer, 1992.

[RD97] M. Reichert and P. Dadam. ADEPTflex - Supporting Dynamic Changes of Workflow without losing control. *Journal of Intelligent Information Systems (JIIS), Special Issue on Workflow and Process Management*, 1997.

[Rei85] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[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.

[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, 1996.

[SH96] P. Straub and C. Hurtado. Business Process Behavior is (Almost) Free-Choice. In *Proceedings of the IMACS-IEEE Multiconference on Computational Engineering in Systems Applications (CESA'96)*, Lille, France, July 1996.

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

[SO99] W. Sadiq and M.E. Orlowska. Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models. In *Proceedings of the 11th Conf on Advanced Information Systems Engineering (CAiSE'99)*, pages 195–209, Hildeberg, Germany, June 1999.

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

[Tau89] Dirk Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, volume 369 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, Germany, 1989.

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

[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.

[Wil77] M. H. Williams. Generating structured flow diagrams: the nature of unstructuredness. *Computer Journal*, 20(1):45–50, 1977.

[Wor99a] Workflow Management Coalition. Interface 1: Process Definition Interchange Process Model. Document Number WfMC TC-1016-P, Document Status - Version 1.1, October 1999. www.wfmc.org.

[Wor99b] Workflow Management Coalition. Terminology & Glossary. Document Number WFMC-TC-1011, Document Status - Issue 3.0, February 1999. www.wfmc.org.