ETH zürich

ETH Library

Detecting, Understanding, and Fixing Control-Flow Errors in Business Process Models

Doctoral Thesis

Author(s):
Favre, Cédric

Publication date:
2014

Permanent link:
https://doi.org/https://doi.org/10.3929/ethz-a-010421900

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

DISS. ETH NO 22266

Detecting, Understanding, and Fixing Control-Flow Errors in Business Process Models

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
Cédric Favre

MSc in Computer Science, EPFL

born on 20.04.1984

citizen of Bavois (VD)

accepted on the recommendation of
Prof. Dr. Peter Müller
Prof. Dr. David Basin
Prof. Dr. Wil M.P. van der Aalst
Dr. Hagen Völzer

2014

Abstract

Business process management targets the continuous optimization of business processes within an organization using information systems. This management discipline uses business process models as central artifacts in numerous use cases including simulation, specification and code generation of an IT solution, or the direct execution of a business process model on a workflow engine.

Business process models frequently contain control-flow errors, such as deadlocks. These errors have severe consequences on the executability of the business process model: A control-flow error can lead to unexpected results, cause runtime exceptions, or completely block the process execution. Being able to detect these errors automatically during the modeling of the process allows companies to save time and money.

A business process is modeled by a graph of activities arranged to achieve a specific business goal. The control-flow of a business process model is defined by the structure of its underlying graph. Consequently, we characterize control-flow errors in terms of structural error patterns, i.e., in terms of three graph structures that are indicative of a control-flow error. We present a technique to detect these error patterns in polynomial time. Upon detecting a control-flow error, the technique delivers to the user a visualization of the error pattern in the business process model and a reduced error trace.

The business processes are modeled using industrial languages which are more complex than the control-flow models used for their analysis. We prove that the control-flow models used by many existing analysis techniques, including the structural approach mentioned previously, cannot represent in a satisfactory manner a particular type of synchronization logic, viz. the inclusive OR-join, used in some popular industrial languages.

We propose a second control-flow analysis technique to check acyclic business process models that may contain inclusive OR-joins. This technique computes the control-flow relationships between the elements of the business process models in quadratic time. We use these relationships to derive a control-flow analysis technique and as a means for the user to reason about the control-flow errors.

Finally, we show how the presented techniques can be combined to obtain a control-flow analysis tool allowing a user without a verification background to perform a control-flow analysis at modeling time. The tool is designed to support the user in detecting, locating, understanding, and fixing control-flow errors. We validate the applicability of this tool on a set of 1350 industrial business process models.

Résumé

Business process management (fr. la gestion des processus commerciaux) vise à continuellement optimiser les processus au sein d'une organisation au moyen de systèmes d'information. Cette discipline de management est basée sur des modèles de processus. Ces modèles ont de nombreuses applications comprenant la simulation, la spécification et la génération d'une solution informatique, ou encore d'être directement exécutés par un moteur de processus.

Les modèles de processus métiers contiennent fréquemment des erreurs de flux de contrôle, comme une deadlock par exemple. Ces erreurs ont de graves conséquences sur l'exécutabilité du modèle: Une erreur de flux de contrôle peut engendrer des résultats inattendus, provoquer des exceptions, ou bloquer l'exécution du processus. La détection précoce d'une erreur de flux de contrôle peut permettre des économies considérables.

Un processus est modélisé par un graphe composé d'activités organisées de manière à atteindre les objectifs de l'entreprise. Le flux de contrôle d'un modèle de processus est défini par la structure de son graphe sous-jacent. Par conséquent, nous caractérisons les erreurs de flux de contrôle en fonction de trois structures d'erreur, c'est dire, trois structures graphiques qui sont indicatives d'erreurs de flux de contrôle. Nous présentons une technique permettant de détecter ces structures d'erreur en temps polynomial. Lorsqu'une erreur de flux de contrôle est détectée, la technique offre deux types d'information: une visualisation de la structure d'erreur et un contre-exemple.

Les processus sont modélisés au moyen de langages industriels qui sont plus complexes que les modèles utilisés pour leur analyse. Nous prouvons que les modèles utilisés par de nombreuses techniques d'analyse existantes, incluant la technique mentionnée plus tôt, ne peuvent pas représenter un type de synchronisation particuler, appellé 'IOR-join', qui est utilisé par certains langages industriels communs.

Nous proposons une seconde technique d'analyse de flux de contrôle adaptée à l'analyse de processus acycliques qui peuvent contenir des 'IOR-joins'. Cette technique calcule les relations de flux de conrôle entre les éléments d'un processus. Ces relations sont utilisées pour obtenir une technique d'analyse et comme outil de raisonnement pour l'utilisateur.

Finalement, nous montrons comment les techniques présentées peuvent être combinées pour obtenir un outil d'analyse de flux de contrôle, qui permet à un utilisateur sans connaissances préalables en vérification d'effectuer une analyse de flux contrôle lors de la modélisation d'un processus. L'outil

est conçu pour assister l'utilisateur lors de la détection, la localisation, la compréhension et la correction d'une erreur de flux de contrôle. Nous validons cet outil sur un ensemble de 1350 modèles de processus.

Acknowledgements

I view my doctoral studies as a professional and personal odyssey. I would like to express my profound gratitude to anyone who positively influenced the course of this journey. You made it possible! In particular, I would like to thank the following people:

Hagen Völzer, my mentor at IBM Research, for supporting me from the first day of my Master thesis to the defense of my PhD. His constructive feedback, positive thinking, and availability have been essential components in shaping this work. I learned a lot and enjoyed working with him.

Prof. Peter Müller for supervising my PhD thesis. His guidance, encouragement, and concrete suggestions were crucial to the completion this thesis. His dedication to research and his way of interacting with his peers have always been inspiring.

Prof. Wil van der Aalst and Prof. David Basin for accepting to review my thesis, providing detailed feedback, and asking challenging questions during the defense.

Prof. Jana Koehler, my former manager, for giving me the opportunity to work at IBM Research and supporting me even long after leaving the company. I also would like to thank the members of the former Business Integration Technologies group for creating a great working environment.

The former and current PreDocs of IBM Research Zurich for providing an inexhaustible source of inspiration and welcomed distraction. Be it during an office discussion, a ski tour, a social event, a climbing session, or simply while sharing a cup of coffee, it was great to share the ebbs and flows of the PhD life with you.

The former and current members of the Chair of Programming Methodology at ETH Zurich for providing such a friendly and stimulating research environment. Your feedback has been instrumental for the development of my thesis.

I am lucky to count many people as my friends and am grateful to all of them for their support. I would like to address a special 'thank you' to Matthias Betschart, Loic Matthey, Michael and Ksenia Wahler, Gregory Neven, Franz-Stefan Preiss, Patrik Bichsel, Jussi Vanhatalo, Ettore Ferranti, Valentin Wüstholz, Roland Häusler, Tamara Ulrich, and Oleksandr Maistrenko for their key support and advice during the last years.

Finally, I would like to thank my family, in particular my parents, Francis and Sabina, and my brothers, Quentin and Niels, for the unconditional

support that they always provided. Last but not least, I would like to thank my wife, Barbara, for sharing the ups and downs of the entire journey with me.

Contents

List of Figures ix

I. Context 1

1. Introduction 3

1.1. Business process management and business process models 3
1.2. Control-flow analysis 6
1.2.1. Control-flow errors 6
1.2.2. Goal of the thesis 7
1.3. State of the art in control-flow analysis of business process models . . . 8
1.4. Open problems 9
1.5. Contributions 10
1.6. Dissertation structure 12

2. Foundations 15

2.1. Basics notions 15
2.2. Petri nets and workflow nets 17
2.2.1. Petri nets 17
2.2.2. Workflow nets 20
2.3. Workflow graphs 22
2.3.1. Acyclic workflow graphs 22
2.3.2. Acyclic workflow graph semantics 23
2.4. Relationship between workflow graphs without IOR and free-choice workflow nets 24

2.5. Control-flow correctness criterions 27
2.5.1. Soundness 27
2.5.2. Alternative correctness criterions 29

II. Control-Flow Analysis **31**

3. Structural and Dynamic Diagnostic Information **33**
3.1. Motivation 33
3.2. Structural errors 34
3.2.1. An existing characterization 34
3.2.2. Siphons 36
3.2.3. Error patterns 37
3.2.3.1. A siphon that does not contain the initial place 38
3.2.3.2. There exists a DQ siphon with a P/T handle 38
3.2.3.3. There exists a simple path to the final place with a T/P handle 40
3.3. Correspondence of the error patterns to an error trace 41
3.3.1. Basic concepts to build an error trace in quadratic time 41
3.3.2. Error trace corresponding to a siphon that does not contain the initial place 47
3.3.3. Error trace corresponding to DQ siphon with a P/T handle . . . 47
3.3.4. Error trace corresponding to a simple path to the final place with a T/P handle 48
3.3.5. Error trace corresponding to an error pattern 49
3.4. A structural characterization of soundness 49
3.5. A polynomial technique to check the structural characterization 53
3.5.1. General flow of the analysis 54
3.5.2. Checking for siphons that are not initially marked 55
3.5.3. Checking that the connected workflow net is structurally sound . 56
3.5.3.1. Identifying minimal siphons 57
3.5.3.2. Checking that a minimal siphon generates a P-component 57
3.5.3.3. Obtaining an error pattern from a failed SMD check . . 58
3.5.4. Checking the rank equation 59
3.5.5. When the rank equation check fails 59
3.5.5.1. An unsound connected workflow net for which a state machine decomposition exists 59
3.5.5.2. Reducing the connected workflow net 60
3.5.5.3. Unsoundness monotonicity 63
3.5.5.4. An unsound SMD connected workflow net is always reduced by Alg. 4 64
3.6. Summary 68

3.7. Related work on structural analysis 68

4. Replacing Inclusive OR logic **71**
4.1. Problem 72
4.2. Translation requirements 73
4.3. Translating IOR-splits 73
4.4. Local replacements for IOR-joins 74
4.4.1. Local replacement and equivalence 75
4.4.2. Characterization of locally replaceable IOR-joins and replacement technique 77
4.4.3. The limitation of local replacements 83
4.5. Non-local replacements for IOR-joins 84
4.5.1. Non-local replacement and equivalence 85
4.5.2. A simple non-local replacement 87
4.5.3. K-replacement 88
4.5.4. A note on complexity 93
4.6. The difficulty of replacing an IOR-join 94
4.7. On the translation to (non-free-choice) Petri nets 96
4.8. Summary 97
4.9. Related work 98

5. Symbolic Execution of Acyclic Workflow Graphs **101**
5.1. Symbolic execution and always-concurrent edges 102
5.1.1. Equivalence of edges and a characterization of deadlock 102
5.1.2. Symbolic execution 105
5.1.3. A normal form for symbols 108
5.1.4. Complexity of the computation 112
5.2. Lack of synchronization and sometimes-concurrent edges 114
5.2.1. Handles and lack of synchronization 115
5.2.2. Detecting handles 119
5.2.2.1. Computing a line graph 120
5.2.2.2. Generating the state graph 120
5.2.2.3. Checking for handles 122
5.2.3. Combining symbolic execution with handle detection 123
5.2.4. Sometimes-concurrent 123
5.3. Dealing with over-approximation 124
5.3.1. User interaction to deal with over-approximation 124
5.3.2. Relaxed soundness 126
5.4. Weak soundness is NP-Hard 127
5.5. Summary 130
5.6. Related work 131

III. Analysis of Industrial Business Process Models 133

6. Building an Analyzer 135
6.1. Modeling the control-flow of a business process . . . 136
6.2. Additional techniques . . . 136
6.2.1. The refined process structure tree . . . 137
6.2.2. Heuristics . . . 139
6.2.3. Kiepuszewski completion . . . 141
6.2.4. State space exploration . . . 143
6.3. Architecture . . . 143
6.4. The choice of the techniques and their priority . . . 145
6.5. The error display and fix suggestion . . . 146
6.5.1. Tool mock up . . . 146
6.5.2. Error display . . . 149
6.5.3. Fix suggestions . . . 150
6.5.4. A note on error dismissal . . . 152
6.6. Tools developed . . . 152
6.6.1. Plugins for IBM WebSphere Business Modeler . . . 153
6.6.2. Validation in IBM WebSphere Business Modeler . . . 153
6.6.3. Validation in IBM WebSphere Integration Developer and IBM WebSphere Process Server . . . 153
6.6.4. A note on the inclusion of structural analysis . . . 154
6.7. Related work . . . 154

7. Evaluation 157
7.1. The data set . . . 158
7.2. Setup . . . 159
7.3. Analysis . . . 159
7.3.1. Efficiency . . . 159
7.3.2. Coverage and consumability . . . 161
7.3.3. Additional remarks . . . 166
7.3.3.1. The type of errors found . . . 166
7.3.3.2. The choice and usefulness of the techniques . . . 166
7.3.3.3. The reduction step of structural analysis was not required 168
7.4. Summary . . . 168

IV. Conclusion 169

8. Conclusion 171
8.1. Summary of the contributions . . . 171
8.1.1. Structural analysis and diagnostic information . . . 171

8.1.2. Identification of a coverage limitation 172
8.1.3. A symbolic execution for acyclic business process models 172
8.1.4. Building a control-flow analyzer 173
8.2. Limitations and directions for future work 173

References **175**

List of Figures

1.1. A business process model. 4
1.2. The business process management lifecycle. 4

2.1. A Petri net. 18
2.2. The incidence matrix of the Petri net illustrated by Fig. 2.1. 19
2.3. An extended-free-choice Petri net. 19
2.4. A workflow graph. 22
2.5. Relation between systems based on workflow graphs (WFG) and free-choice (FC) workflow nets (WFN). 25
2.6. Translating nodes of a workflow graph to corresponding Petri net patterns. 26

3.1. A circuit (dashed line), a handle (dotted line), and a bridge (plain line). 34
3.2. A sound connected workflow net containing interesting circuits, handles, and bridges. 35
3.3. A workflow net with a siphon that does not contain the initial place. . . 38
3.4. A DQ siphon with a P/T handle. 39
3.5. A simple path to f with a T/P handle. 40
3.6. Illustration of S and its marked border B. 43
3.7. Illustration of S and its marked border B after σ_1. 43
3.8. Illustration of S_2 and its marked border B_2. 44
3.9. A Circuit with a T/P handle. 51
3.10. Path with T/P handle case 1. 52
3.11. Path with T/P handle case 2. 52
3.12. The general flow of the analysis algorithm. 55
3.13. A SMD connected workflow net containing a deadlock. 60

3.14. The result of the two reduction functions on the connected workflow net $\overline{N}$ illustrated by Fig. 3.13. 61
3.15. Illustration for the proof of Lemma 3.30. 65
3.16. The path H^* is a T/P handle of Ω^*. 67

4.1. Two possible ways to replace an IOR-split. 74
4.2. A faithful quadratic translation of an IOR-split creating multiple sinks. 75
4.3. An example of local replacement where the IOR-join j is replaced by the partial workflow graph composed of the nodes v and w, and the edge i. 75
4.4. Canvas of cover-based replacement. 78
4.5. An example of local replacement where the IOR-join j is replaced by the partial workflow graph composed of the nodes w, x, y, and z and the edges f, g, h, and i. 79
4.6. Two workflow graphs containing an IOR-join that cannot be replaced locally. 84
4.7. An example of non-local replacement. 85
4.8. Non-local replacements of the IOR-joins in Fig. 4.6 that do not have a local replacement. 87
4.9. Non-local replacement of j in Fig. 4.7(a). 88
4.10. The IOR-join y cannot be replaced. 93
4.11. Prefix that cannot be completed. 95
4.12. A non-free-choice Petri net that is equivalent to the one in Fig. 4.10. . 97

5.1. A workflow graph. 103
5.2. The assignment resulting from the symbolic execution of the workflow graph of Fig. 2.4. 105
5.3. The propagation rules. 106
5.4. Display of a deadlock. 112
5.5. A workflow graph that contains a lack of synchronization. 115
5.6. The line graph of Fig. 2.4. 120
5.7. The state graph of Fig. 5.6. 121
5.8. A portion of the state graph for the line graph in Fig. 5.6. 122
5.9. A deadlock. 124
5.10. A lack of synchronization. 125
5.11. A deadlock and a lack of synchronization. 125
5.12. A deadlock located at J that should not be dismissed. 126
5.13. A lack of synchronization that should not be dismissed. 127
5.14. The workflow graph obtained for the formula $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$. 128

6.1. An example of workflow graph. 137

6.2. Decomposition of the workflow graph of Fig. 6.1 using the Refined Process Structure Tree parsing. 138
6.3. The most complex fragments of Fig. 6.2 that can be analyzed in isolation. 139
6.4. A worklfow graph with multiple sinks. 142
6.5. The worklfow graph of Fig. 6.4 completed to obtain a single sink. 142
6.6. Architecture of our control-flow analyzer. 144
6.7. Errors appear on the canvas during the modeling phase. 147
6.8. Displaying an error trace for the deadlock. 147
6.9. Displaying an error trace for the lack of synchronization. 148
6.10. Displaying the conditions leading a trace into a lack of synchronization. 149

7.1. Screenshot of the display of a siphon that is not initially marked. 162
7.2. Screenshot of the display of a simple path to the final place with a T/P handle. 163
7.3. Screenshot of the display of a DQ-siphon with a P/T handle. 163
7.4. Screenshot of the display of an error trace obtained using symbolic execution. 165
7.5. Screenshot of the display of a simple path to the final place with a T/P handle due to a wrongly specified instantiation semantics. 167
7.6. Screenshot of the display of a siphon that is not initially marked displayed in a portion of a large process. 167

Part I.

Context

1

Introduction

In this chapter, we introduce the context and contributions of this thesis. We first present the discipline of business process management and its central artifact: business process models. Then we discuss the problem of control-flow errors in business process models and the existing analysis techniques to detect, locate and understand them. Finally, we present the contributions of this thesis to the control-flow analysis of business process models.

1.1. Business process management and business process models

Business Process Management (BPM) is a management discipline centered around business processes. It targets to continuously increase the effectiveness and efficiency of business processes within an organization using information systems (61). Business processes span organizational boundaries, linking together people, information flows, systems and other assets to create and deliver value to customers and constituents (29).

BPM makes the business processes explicit using, most of the time, graphical business process models. A *business process model* is composed of an ordered collection of activities organized to achieve a specific business goal. The activities within a business process model represent various types of tasks including sub-processes, automated tasks, and tasks carried out by humans. Fig. 1.1 illustrates a business process model describing the handling of an order. For the sake of presentation, this example is simplified.

Activities are represented by rectangles. The routing logic is defined by the nodes with a diamond shape: An empty diamond expresses a decision, i.e., one of the dia-

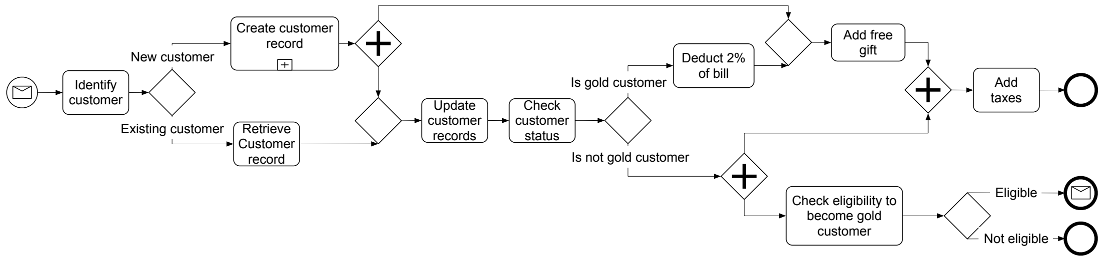


Figure 1.1.: A business process model.

mond's outgoing flow is executed. An empty diamond with multiple incoming edges is used to join alternative flows. A diamond with a plus inside expresses that all its outgoing flows are executed concurrently. A diamond with a plus sign and multiple incoming edge is used to join concurrent flows.

The model illustrated by Fig. 1.1 describes the processing of an order: First the customer is identified. Then a decision is made based on whether the customer is a new or an existing customer. For an existing customer, his customer record is retrieved. For a new customer, a subprocess, represented by the activity with the little plus, is invoked in order to create a new customer record. Later in the process, it is evaluated on whether the customer is a gold customer or not. For gold customers, there is a deduction of 2% on the bill and a free gift. New customers also get a free gift.

BPM undertakings are usually organized into several phases such as design, implementation, enactment, monitoring and analysis (10, 86). The cycle comprising these phases is referred to as *business process management lifecycle* (see Fig. 1.2).

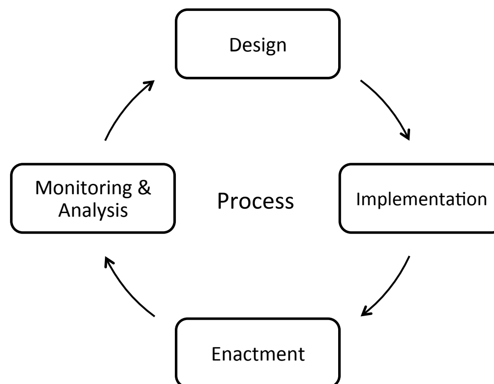


Figure 1.2.: The business process management lifecycle.

The BPM lifecycle suggests a continuous improvement methodology that can be started from any of its four phases. Business process models play a central role in the business process management lifecycle: During the design phase, new business process

models are (re-)designed based on requirement analysis. In the implementation, the business process models are refined or translated into an IT solution. The solution can then be enacted, i.e., deployed and executed. The running solution is monitored and its observation may reveal problems and aspects that can be improved.

Over the years, IT vendors have built integrated BPM suites supporting the entire BPM lifecycle. Such BPM suites support the development of IT solutions following a *model driven approach* (62) in which the business process models are the main artifacts and are shared across the different phases of the BPM lifecycle. These shared business process models foster the communication and collaboration between business stakeholders, such as process owner or business analysts, and IT stakeholders, such as IT architects and developers. The use cases for business process models in BPM suites extend across the entire BPM lifecycle:

Documentation: More and more companies model their business processes. As main artifacts of the design phase, these process models are not only the basis of further use cases such as analysis, simulation, or direct execution but also a representation, shared between business and IT practitioners, of the business logic.

Analysis: Various static analysis techniques, such as cost or throughput analysis, can be used to better understand a process, its bottlenecks, and optimization potential.

Simulation: Instead of statically analyzing a process, various vendors offer tools allowing its user to get a better understanding of the process dynamic behavior via a step through animation of the process execution.

Generation of the implementation: A business process model can serve as blueprint for the generation of an IT solution. The level of automation in the generation of the IT-solution depends on the refinement of the business process model. Typically, the flow of control of the application, i.e., the order in which the activities are executed can be generated automatically. The automated activities themselves are typically implemented by services following a *service oriented architecture (SOA)*.

Execution on a process engine: As alternative to the generation of an implementation, modern BPM suites include a *process engine* that can execute a business process model directly.

Monitoring: Running processes can be monitored, i.e., metrics, called *key performance indicators (KPI)*, are gathered during the execution of the processes and are made available to IT operators and business analysts. Examples of such metrics include the average time to complete a process or the number of processes currently running. KPIs are used by IT operators to assess the state of the system: For example to monitor that a *service level agreement (SLA)* is satisfied. Analysts can also use KPIs to identify problems in the process.

Optimization: Based on analysis, simulation, or monitoring results, potential improvements of a process are identified and give rise to a redesign.

1.2. Control-flow analysis

In 1976 already, Boehm pointed out that defects identified early require less time and money to fix than defects identified at later stage (3). Transferred to software solutions built using BPM suites, this observation has the direct consequence that errors in business process models should be detected as early as possible, i.e., during the design phase. This thesis focuses on detecting, identifying, and fixing control-flow errors in business process models during their modeling.

1.2.1. Control-flow errors

Numerous studies have shown that, in practice, business process models frequently contain control-flow errors (see the work of Mendling (50) for a survey).

Control-flow errors arise because of slips during the modeling activity, misunderstanding of the semantics of the modeling language, incomplete understanding of the process being modeled, or the process becoming too complex for the modeler to maintain an understanding of the control-flow.

Let us get back to the business process model of Fig. 1.1. Consider an execution where an existing customer is not yet a gold customer. In such an execution, only the second incoming flow of the right-most diamond with a plus sign will be executed. Therefore, the diamond with a plus sign will not execute because it synchronizes all of its incoming flows. This will create a *deadlock* preventing the execution of the activity 'Add taxes' and the completion of the process. Beside deadlocks, there exist a second type of control-flow error called *lack of synchronization*. A lack of synchronization occurs when two, or more, activated flows are not synchronized properly. In the business process model of Fig. 1.1, a lack of synchronization would occur when a new customer is also a gold customer. In such an execution, the empty diamond following the activity labeled 'deduct 2% of bill', which role is to merge all incoming flows without synchronizing them, would be executed twice. Any element following the empty diamond would potentially also be executed twice.

Control-flow errors in a business process model have severe consequences on its executability: A control-flow error can lead to unexpected results, cause runtime exceptions, or completely block the process execution. The negative consequences of a deadlock are clearly recognizable: a deadlock prevents an execution to continue. The effects of lack of synchronization are more pernicious: while, in the example above, it does not sound dramatic to send two free gifts instead of one, the consequences of repeating activities multiple times might be severe and are often unintended by the author of the modeler of the process. It is also difficult for a modeler to cope consistently with executions using multiple concurrent active flows on the same path. Finally, lack

of synchronization are not supported by all modeling languages. For example, executing a model using the *Business Process Execution Language (BPEL)* (39), an exception would be thrown in case of lack of synchronization.

When using business process models in simulation, code generation, and execution, it is crucial that the processes are free of control-flow errors. Moreover, a process containing a control-flow error is an unfaithful model: it unlikely that the modeler intended the process to be blocked or to execute some activities twice. Therefore, finding and fixing a control-flow error is always relevant, even when a business process model is "only" used for documentation purposes.

1.2.2. Goal of the thesis

The goal of this thesis is to develop a control-flow analysis technique and a tool allowing a user without a verification background to detect, locate, understand, and correct a control-flow error in a business process model.

In the context of this thesis, a user of a business process modeling tool is referred to as a modeler. This modeler can carry various roles in its organization such as business analyst or IT architect for example. The background and focus of a modeler also varies depending if he is a business or a technical expert. We will assume a modeler to have a basic understanding of the semantics of the elements composing a business process and the possible control-flow errors, but not necessarily to have a verification background.

We aim to build an analyzer which supports the modeler in designing a business process free of control-flow errors. Such analyzer and therefore its underlying analysis technique(s) should meet the following requirements:

Efficiency: Analyzing the process during the modeling phase requires an analysis technique that is efficient enough to avoid slowing down the modeling activity.

Consumability: Upon detecting an error, the technique should produce diagnostic information that is consumable, i.e., allow the modeler to locate, understand, and fix the error.

Coverage: Business process modeling languages offer a wide range of modeling constructs. The technique should be able to analyze industrial business models containing the largest set of useful[1] constructs as possible.

[1] Recent business process modeling languages, in particular BPMN 2.0 (54), have been criticized for being too complex, i.e., defining too many constructs. BPMN 2.0 is aimed to be an executable language. Therefore, the complete standard is fairly complex, probably too complex for many types of users. Practitioners, such as Bruce Silver (60), suggest that the different types of user should use different subsets of the language. Discussing if the complexity of BPMN 2.0 is justified by sufficient benefits for the users, or if only a subset of the language should be used, is beyond of scope of this thesis.

We will discuss, in the next sections, that many control-flow analysis techniques have been exist in the literature but that none of them meets all of these three requirements simultaneously.

1.3. State of the art in control-flow analysis of business process models

In this section, we present a landscape of control-flow analysis techniques available at the beginning of this work. We will compare in more details our contributions with the related work at the relevant locations of this thesis.

Typically, the control-flow component of a business process model is translated into a simple control-flow model such as a *workflow graph* (57, 74) or a *workflow net* (64, 67). Such a translation decouples the analysis techniques, applied subsequently, from a particular business process modeling language and allows analysis technique designers to cope with the complexity and specificities of the business process modeling languages. The absence of control-flow errors in a workflow graph or a workflow net is called *soundness*.

Existing analysis techniques vary in terms of efficiency, coverage of the various business process modeling language constructs, and consumability. Factoring out for a moment the coverage requirements, we observe a trade-off between efficiency and consumability:

On the one hand, there exist efficient techniques that are based on the theory of *free-choice Petri net*, a class of Petri nets that can express the control-flow of a business process model (67). The most efficient known technique, which is based on the so-called *rank theorem*, can determine the soundness of a free-choice workflow net in cubic time (8, 9, 53). Unfortunately, this technique does not provide any useful diagnostic information. Esparza (13) provides a set of reduction rules for free-choice Petri nets. The Petri net is sound if and only if it can be reduced to the empty net. These reductions allow one to verify soundness in polynomial, but more than cubic, time. Although this approach might give rise to some diagnostic information, it has not been worked out yet. Similarly, Sadiq and Orlowska (57) propose reduction rules for workflow graphs. These rules are not complete, i.e., cannot reduce some sound workflow graphs.

On the other hand, *state space exploration*, i.e., the exhaustive search of the control-flow state space of a business process model, provides an execution sequence, called error trace, that exhibits an erroneous behavior of the workflow graph as diagnostic information. The error trace is currently the most consumable diagnostic information. State space exploration is also a more versatile approach, which can deal with non-free-choice models. Unfortunately, in the worst case, state space exploration has an exponential time complexity. State space exploration has been exploited as core or complementary analysis technique by various authors (52, 77, 83). In some cases, state space exploration was used to check additional control-flow properties beyond

soundness.

To mitigate the impact on the execution time of high complexity techniques such as state space exploration, Zerguini (85) proposes to decompose a Petri net into fragments that can be analyzed in isolation. The approach requires a quadratic time to decompose a Petri net. Alternatively, Vanhatalo et al. (56, 73, 74) present two linear time complexity decomposition techniques and a set of heuristics allowing to determine the soundness of *some* fragments in linear time. Other approaches have been used to reduce the size of the model such as the reduction rules from Murata (53) by the authors of the control-flow analysis tool Woflan (77) or partial order reduction techniques (44) by the authors of the tool LoLA (82). In a case study where the diagnostic information was not considered, we observed that a combination of state space exploration and decomposition or reduction techniques allows us to determine efficiently the soundness of some industrial business process models (19).

1.4. Open problems

In this section, we position ourselves on the onset of this work to describe the open problems of the control-flow analysis techniques described in the literature:

No polynomial analysis technique providing diagnostic information: Existing approaches present a trade-off between efficiency of the technique and the quality of the diagnostic information that it provides (74). Even for free-choice Petri nets, for which efficient analysis techniques exist in the literature, It is an open question whether diagnostic information can be obtained in polynomial time or not.

Insufficient diagnostic information: Even the error trace, which is currently the most eloquent and vivid description of a control-flow error, has some drawbacks: Error traces are often long, and only a portion of the error trace might be relevant to understand the error. Many authors have realized that, independently of the efficiency requirements, progress needs to be made in providing better diagnostic information (18, 50). Whether we can improve the diagnostic information beyond the error trace is an open question.

False positives due to the abstraction of data-based decisions: Aside from the discussion around the efficiency and consumability, existing techniques also present the following limitation: control-flow analysis techniques abstract from the data-constraints influencing the decisions of the process. In the context of the process of Fig. 1.1 for example, it is impossible that a customer is a new customer and a gold customer at the same time. Such a constraint, which restricts the number of actual executions of the process, is not taken into account by any existing analyzer. Such abstractions can result in false positives, i.e., detecting control-flow errors that will never occur in practice. The approaches in the literature do not allow a user to prevent or deal with these false-positives.

Coverage limitations: The efficiency of most control-flow analysis techniques, is obtained via the use of simple formalisms, such as workflow nets. In practice however, business processes are usually modeled with industrial languages such as BPMN 2.0 (54), which offer a plethora of control-flow constructs. While many translations from business modeling language to simpler formalism exist, complete translations are rare and often complex, in term of the resulting model. Such complex translations have an impact on the efficiency of the technique and the consumability of its diagnostic information. Some of the complex control-flow constructs, such as the *inclusive OR (IOR)* routing logic for example, are only covered by a few of the existing control-flow analysis techniques. We believe that some coverage limitations are unrecognized.

1.5. Contributions

This dissertation presents two novel control-flow analysis techniques that provide new means of understanding and dealing with control-flow errors. Both techniques have a polynomial time worst-case complexity. Furthermore, we prove the existence of coverage limitations for existing efficient control-flow analysis techniques.

The first analysis technique is a structural analysis, i.e., the analysis as well as the diagnostic information are based on the structure of the business process model. The control-flow of a business process model is directly captured by the structure of its underlying graph. Control-flow errors themselves are generated by recognizable graph structures, which we call *error-patterns*. Highlighting such an error-pattern in the business process model provides natural and comprehensive diagnostic information. Following this line, we characterize soundness as the absence of three structural error-patterns and present a control-flow analysis technique based on this characterization. In case of an unsound model, the technique provides two types of diagnostic information: 1. A structural error-pattern that can be displayed on the modeling canvas and allows the modeler to understand the structure responsible for the control-flow error. 2. An error trace that exemplifies an improper behavior of the process resulting from this error-pattern. Both, the analysis and the derivation of the two types of diagnostic information, have polynomial worst-case complexity.

Our second main contribution shows a coverage limitation. The business processes are modeled using industrial languages, which are significantly more complex than the control-flow models used for their analysis. The coverage of an analysis technique is directly linked to how well a business process model control-flow can be translated into the control-flow model on which the technique is based. In this context, we prove that processes containing one routing element, viz. the *IOR-join*, cannot always be translated into *free-choice Petri nets* in a satisfactory manner. The existing efficient analysis techniques rely on the control-flow being modeled by a free-choice Petri net or an equivalent formalism. Therefore, this result shows an important coverage issue

of many existing control-flow analyzers, including the structural analysis presented in this thesis. As a by-product of the study of these coverage limitations, we describe two IOR-join translation techniques and characterize their application conditions.

The second analysis technique is called *symbolic execution*. It allows us to decide the soundness of an acyclic business process model, which may contain IOR-joins. It also delivers a new type of diagnostic information and has a quadratic worst-case time complexity. Symbolic execution computes equivalent sets of executions, with respect to the elements being executed, to obtain the control-flow relationships between the elements of the business process model. For example, in the process illustrated by Fig. 1.1, symbolic execution establishes that the activities '*Retrieve customer record*' and '*Create new customer record*' are alternative, i.e., are never executed concurrently and that the activities '*Add a free gift*' and '*Check eligibility to become a gold customer*' are sometimes executed concurrently. Such control-flow relationships can be used for various applications including data-flow analysis or the comparison of processes. We use these relationships to obtain a control-flow analysis technique. The conditions themselves allow the modeler to reason about the error in a novel way and to efficiently deal with spurious errors that arise due to over-approximation of the data-based decisions in the process.

As a side result, we use symbolic execution to prove that deciding an alternative control-flow correctness criterion, called *weak soundness*, of an acyclic business process model containing IOR-joins is NP-hard.

While the techniques presented earlier have a polynomial worst-case complexity and provide a reasonable diagnostic information, the structural analysis cannot deal with processes containing IOR-joins and the symbolic execution can only deal with acyclic process models. To overcome these individual coverage limitations, we have built a hybrid control-flow analyzer which combines the techniques developed in this thesis. We validate this analyzer against industrial business process models thus showing that a hybrid analyzer can satisfy the requirements of Sect. 1.2.2.

In summary, this thesis presents the following contributions:

1. A new structural characterization of soundness.
2. A structural analysis technique providing structural and exemplified diagnostic information in polynomial time.
3. A proof that IOR-joins cannot always be translated into free-choice Petri nets in a satisfactory manner.
4. Two techniques allowing us to replace many IOR-joins by free-choice Petri nets.
5. A new technique, called symbolic execution allowing us to analyze the control-flow of an acyclic business process models containing IOR-joins in quadratic time.

6. A novel diagnostic information, for acyclic processes, allowing a user to obtain the control-flow relationship between the elements of the process and to reason in terms of equivalent sets of executions.

7. An approach, for acyclic processes, allowing a user to dismiss false positives arising from the data abstraction occurring in existing control-flow analyses.

8. A proof that deciding weak soundness of acyclic process models containing IOR-joins is NP-hard.

9. A strategy combining the techniques mentioned earlier to obtain a control-flow analyzer satisfying the requirements described in Sect. 1.2.2. This strategy was implemented and validated on more than 700 industrial business process models.

Some of these contributions have been published as peer reviewed conference, journal, and workshop papers: The results on the translation of IOR-logic (Contributions 3 and 4) have been presented at the 10th International Conference on Business Process Management (27) and an extended version of the results was published in the Information System journal (22). The first insights leading to the symbolic execution technique (contributions 5, 6, and 7) were presented at the 2$^{\text{nd}}$ Central-European Workshop on Services and their Composition (21). The complete results have been presented at the 8th International Conference on Business Process Management (26). Additionally, a patent (28) on these results is pending.

We integrated various combinations of the developed techniques into numerous IBM products (35, 36, 37, 38). Prototypical versions of our implementations were also made available as tools or discussed in publications: A case study evaluating, among two other tools, a first version of the hybrid analyzer was presented at the 7th International Conference on Business Process Management (BPM09) (19) and an extended version of the study published in the Data & Knowledge Engineering journal (19). We also demonstrated this version control-flow analyzer in the demonstration track of BPM09 (24). A later version integrating the symbolic execution technique was made available as part of a set of plugins of IBM WebSphere Business Modeler (36) and described in a series of articles on IBM developerWorks (34). Another similar version was made available as a service (23).

1.6. Dissertation structure

The presentation of the contributions discussed earlier is organized as follows:

In Chap. 2, we introduce formally fundamental concepts such as the control-flow models used in this thesis, the related notions of correctness, and the relationship between these models.

In Chap. 3, we present our structural characterization of soundness and the corresponding control-flow analysis technique.

In Chap. 4, we show the difficulty of translating IOR-joins and present some techniques, with their conditions of applicability, to translate an IOR-join.

In Chap. 5, we present the symbolic execution technique, the diagnostic information that it provides, and show how it allows a user to efficiently deal with spurious errors that arise due to over-approximation of the data-based decisions in the process. Finally, we prove that deciding weak soundness for an acyclic workflow graph containing IOR-joins is NP-hard.

In Chap. 6, we present a control-flow analyzer satisfying the efficiency, consumability, and coverage requirements discussed in Sect. 1.2.2. We also discuss the user interaction that the analyzer provides and the different versions of this analyzer which have been transferred into IBM products.

In Chap. 7, we evaluate the performances and suitability of the analyzer presented in Chap. 6 on industrial business process models.

In Chap. 8, we summarize the contributions of this dissertation and discuss possible future work.

2

Foundations

In this chapter, we present the two standard abstractions that are used to represent the control-flow of a business process model in this thesis: *workflow nets* and *workflow graphs*. Workflow nets are a kind of Petri nets that are commonly used to represent the control-flow of a business process model and allow us to leverage existing analysis approaches developed for Petri nets. Workflow graphs have been the first abstraction used in the literature (57) for the analysis of business process models, they also have a one to one mapping with the main control-flow constructs of industrial modeling languages. Last but not least, workflow graphs, as we will define them, include the *inclusive or-logic* which is part of many industrial modeling languages but is not represented easily in workflow nets as we will discuss in Chap. 4.

Workflow nets and workflow graphs are graphs. In Sect. 2.1, we define basics notions including the fundamental notions related to graphs and the notion of multi-sets. In Sect. 2.2, we define workflow nets and their semantics. In Sect. 2.3, we define workflow graphs and their semantics. In Sect. 2.4, we discuss the relationship between workflow nets and workflow graphs (without inclusive or-logic). In Sect. 2.5, we define the notions of control-flow correctness that we use in this thesis.

2.1. Basics notions

A workflow net or a workflow graph is represented by a graph or a multi-graph respectively. The difference between a graph and a multi-graph is that in a multi-graph edges are first class citizens which can be referred to whereas, in a graph, the edges (in the following called arcs) are relations between nodes. While graphs are sufficient to define workflow nets, multi-graphs are necessary to the definition of workflow graphs and

their semantics because, as we will see later, we need the capability to differentiate two edges between a pair of nodes. We now define the notions of a graph and a multi-graph formally:

Definition 2.1 (Directed graph)**.** *A* directed graph $G = (N, E)$ *consists of a set* N *of* nodes *and a set* E *of ordered pairs* (s, t) *of nodes, called* arcs *and written* $s \to t$*.*

Definition 2.2 (Directed multi-graph)**.** *A* directed multi-graph $G = (N, E, c)$ *consists of:*

- *a set* N *of* nodes*,*
- *a set* E *of* edges*, and*
- *a mapping* $c : E \to (N \cup \{null\}) \times (N \cup \{null\})$ *that maps each edge to an ordered pair of nodes or null values.*

If c *maps* $e \in E$ *to an ordered pair* $(s, t) \in N$*, then* s *is called the* source *of* e*,* t *is called the* target *of* e*,* e *is an* outgoing *edge of* s*, and* e *is an* incoming *edge of* t*.*

If $s = null$*, then we say that* e *is a* source *of the graph. If* $t = null$*, then we say that* e *is a* sink *of the graph.*

For a node $n \in N$*, the set of incoming edges of* n *is denoted by* $\circ n$*. The set of outgoing edges of* n *is denoted* $n\circ$*. If* n *has only one incoming edge* e*,* $^\circ n$ *denotes* e *(*$\circ n$ *would denote* $\{e\}$*). If* n *has only one outgoing edge* e'*,* n° *denotes* e'*.*

In the following we omit to specify that the graphs and multi-graphs are directed. We now define other basic notions for graphs and multi-graphs such as the notion of path or of strongly connected multi-graph. While we define these notions for multi-graphs, they naturally carry over to graphs.

Definition 2.3 (Path)**.** *A* path $p = \langle x_0, ..., x_n \rangle$ from *an element* x_0 to *an element* x_n in *a multi-graph* $G = (N, E, c)$ *is an alternating sequence of elements* x_i *in* N *and in* E *such that, for any element* $x_i \in E$ *with* $c(x_i) = (s_i, t_i)$*, if* $i \neq 0$ *then* $s_i = x_{i-1}$ *and if* $i \neq n$ *then* $t_i = x_{i+1}$*.*

If x *is an element of a path* p *we say that* p contains x*.*

A path is trivial*, if it is contains only one element.*

A path is simple*, if all of its elements are pairwise disjoint.*

A cycle *is a path* $b = \langle x_0 \ldots x_n \rangle$ *such that* $x_0 = x_n$ *and* b *is not trivial.*

Definition 2.4 (Strongly connected)**.** *A multi-graph* $G = (N, E, c)$ *is* strongly connected *if for each pair* (e_1, e_2) *of elements in* $N \cup E$*, there exist a simple path from* e_1 *to* e_2*.*

We now define the notion of multi-set and their related operations:

Definition 2.5 (Multi-set)**.** *Let U be a set. A* multi-set *over U is a mapping $m : U \rightarrow \mathbb{N}$.*

We write $m[e]$ instead of $m(e)$.

For two multi-sets m_1, m_2, and each $x \in U$, we have : $(m_1 + m_2)[x] = m_1[x] + m_2[x]$ and $(m_1 - m_2)[x] = m_1[x] - m_2[x]$.

The scalar product *is defined by $m_1 \otimes m_2 = \sum_{x \in U} (m_1[x] \times m_2[x])$.*

By abuse of notation, we sometimes use a set $X \subseteq U$ in a multi-set context by setting $X[x] = 1$ if $x \in X$ and $X[x] = 0$ otherwise.

2.2. Petri nets and workflow nets

We first define Petri nets and then present workflow nets. Both notions and their definitions are standard from the literature (see (9, 65))

2.2.1. Petri nets

A Petri net is a directed graph containing two types of nodes: places and transitions. More formally:

Definition 2.6 (Petri nets)**.** *A* Petri net *is a triple $N = (P, T, F)$ where:*

- *$P = \{p_1, \ldots, p_i\}$ is a finite set of* places*,*
- *$T = \{t_1, \ldots, t_j\}$ is a finite set of* transitions*,*
- *$P \cap T = \emptyset$, and*
- *$F \subseteq (P \times T) \cup (T \times P)$ is the flow relation. An element of F is called an* arc*.*

Fig. 2.1 depicts a Petri net, places are represented by circles and transition by vertical bars. For now, the reader should ignore that some arcs, places, and transition are highlighted.

We transfer the properties of graphs to Petri nets. For example, a Petri net is strongly connected if its underlying graph is strongly connected. We henceforth use the following usual notations (c.f. (9)):

Definition 2.7 (Petri net properties)**.** *Let $N = (P, T, F)$ be a Petri net.*

1. *The places and transitions of N are also called* nodes*.*
2. *The characteristic function of F is denoted χ_F.*
3. *The matrix $C = \|\ c_{ij}\ \|$ such that: $1 \leq i \leq\mid P \mid$, $1 \leq j \leq\mid T \mid$, and $c_{ij} = \chi_F(t_j, p_i) - \chi_F(p_i, t_j)$ is called the* incidence matrix *of N.*

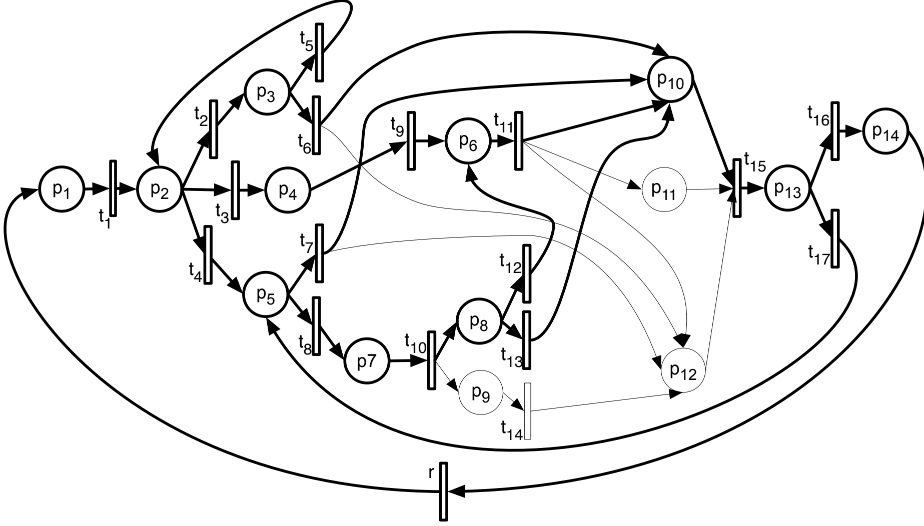


Figure 2.1.: A Petri net.

4. *Let $x \in P \cup T$. The* preset *of x in N, denoted $pre_N(x)$,is given by $pre_N(x) = \{y \in P \cup T \mid (y, x) \in F\}$. The* postset *of x, denoted $post_N(x)$, is given by $post_N(x) = \{y \in P \cup T \mid (x, y) \in F\}$. If the parameter N is clear from the context, we use the notations $\circ x$ instead of $pre_N(x)$ and $x\circ$ instead of $post_N(x)$.The preset (postset) of a set of nodes is the union of presets (postsets) of its elements.*

5. *N is a* P-graph *if for all $t \in T, \mid \circ t \mid = 1 = \mid t\circ \mid$.*

6. *A set $S \subseteq P$ is a* siphon (a trap) of *N if $S \neq \emptyset$ and $\circ S \subseteq S\circ$ ($S\circ \subseteq \circ S$). A siphon (trap) is* minimal *if it does not contain a siphon (trap) as proper subset.*

7. *A Petri net $N' = (P', T', F')$ is a* subnet *of N, denoted $N' \subseteq N$, if $P' \subseteq P$, $T' \subseteq T$, and $F' = F \cap ((P' \times T') \cup (T' \times P'))$. A subnet $N' \subseteq N$ is* generated by *a set S of places of N if $P' = S$ and $T' = \circ S \cap S\circ$.*

8. *$N' \subseteq N$ is a* P-component *of N if N' is a strongly connected P-graph and $T' = \circ P' \cup P'\circ$.*

9. *N is* state machine decomposable (SMD) *or* covered by P-components *if every node of N belongs to a P-component.*

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	r
p_1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
p_2	1	−1	−1	−1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
p_3	0	1	0	0	−1	−1	0	0	0	0	0	0	0	0	0	0	0	0
p_4	0	0	1	0	0	0	0	0	−1	0	0	0	0	0	0	0	0	0
p_5	0	0	0	1	0	0	−1	−1	0	0	0	0	0	0	0	0	1	0
p_6	0	0	0	0	0	0	0	0	1	0	−1	1	0	0	0	0	0	0
p_7	0	0	0	0	0	0	0	1	0	−1	0	0	0	0	0	0	0	0
p_8	0	0	0	0	0	0	0	0	0	1	0	−1	−1	0	0	0	0	0
p_9	0	0	0	0	0	0	0	0	0	1	0	0	0	−1	0	0	0	0
p_{10}	0	0	0	0	0	1	1	0	0	0	1	0	1	0	−1	0	0	0
p_{11}	0	0	0	0	0	0	0	0	0	0	1	0	0	0	−1	0	0	0
p_{12}	0	0	0	0	0	1	1	0	0	0	1	0	0	1	−1	0	0	0
p_{13}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	−1	0
p_{14}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	−1

Figure 2.2.: The incidence matrix of the Petri net illustrated by Fig. 2.1.

Let N be the Petri net illustrated by Fig. 2.1. The incidence matrix of N is given in Fig. 2.2. The set of places $S = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_{10}, p_{13}, p_{14}\}$ is a siphon of N. The siphon S is also a minimal siphon of N. The subnet N' generated by S and highlighted in Fig. 2.1 is a P-component of N. Consider the transition t_{10}, the preset of t_{10} is $\text{pre}_N(t_{10}) = \circ t_{10} = \{p_7\}$, $\text{post}_N(t_{10}) = t_{10}\circ = \{p_7, p_8\}$, and $\text{post}_{N'}(t_{10}) = \{p_7\}$.

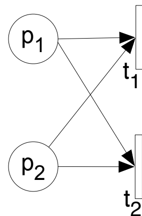


Figure 2.3.: An extended-free-choice Petri net.

A Petri net N is *free-choice* if for any place p of N such that $|p\circ| > 1$, we have for each $t \in p\circ$, $|\circ t| = \{p\}$. A Petri net $N = (P, T, F)$ is *extended-free-choice* if for each place $p \in P$ and each transition $t \in T$, when $(p, t) \in F$ then $\circ t \times p\circ \subseteq F$. The Petri net illustrated by Fig. 2.1 is free-choice. The Petri net illustrated by Fig. 2.3 is extended-free-choice but not free-choice. We call this particular Petri net the *extended-free-choice pattern*. In the rest of the thesis, we only consider free-choice Petri nets and omit to specify it. A known and simple transformation allows us to transform an extended-free-choice Petri net into a free-choice Petri net. We describe this transformation as part of the 'normalization' step in Sect. 2.4.

We will use later the notion of cluster to decompose a Petri net:

Definition 2.8 (Cluster)**.** *Let x be a node of a Petri net N.*

The cluster *of x, denoted by $[x]$, is the minimal set of nodes such that the three following conditions hold:*

1. $x \in [x]$,
2. *if a place p belongs to $[x]$ then $p\circ \subseteq [x]$, and*
3. *if a transition t belongs to $[x]$ then $\circ x \subseteq [x]$.*

The sets $\{p_1, t_1\}$, $\{p_2, t_2, t_3, t_4\}$, and $\{p_{10}, p_{11}, p_{12}, t_{15}\}$ are three clusters of the Petri net illustrated by Fig. 2.1. The Petri net illustrated by Fig. 2.3 has a single cluster containing all of its elements. For any Petri net N, the set of the clusters of N is a partition of the nodes of N (9, Proposition 4.5).

Note that, although we use standard notations for clusters and multi-sets, there is clash between these two notations. If the context is unclear, we will specify explicitly whether we deal with a cluster or a multi-set.

We now define the semantics of Petri nets:

Definition 2.9 (Marking, enablement, and execution sequence). *Let $N = (P, T, F)$ be a Petri net and $S \subseteq P$.*

A marking $m : P \to \mathbb{N}$ *of N is a multi-set over P.*

A place $p \in P$ is marked *by a marking m if $m(p) \geq 1$. When m is clear, we omit to specify it.*

S is marked *if some place of S is marked. The total number of tokens on S is denoted by $m(S)$.*

A marking m enables *a transition $t \in T$ if m marks every place in $\circ t$. A transition t enabled by m can* execute *and its execution leads to the marking m', denoted $m \xrightarrow{t} m'$, such that:*

$$m'(p) = \begin{cases} m(p) + 1 & \text{if } p \in t\circ \text{ and } p \notin \circ t, \\ m(p) - 1 & \text{if } p \in \circ t \text{ and } p \notin t\circ, \\ m(p) & \text{otherwise.} \end{cases}$$

If $m \xrightarrow{t_1} m_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} m_n$ are transition executions, then $\sigma = \langle t_1 \dots t_n \rangle$ is an execution sequence*, written $m \xrightarrow{\sigma} m_n$. A marking m' is* reachable *from m, written $m \xrightarrow{*} m'$ if there exist an execution sequence σ such that $m \xrightarrow{\sigma} m'$. An execution sequence can be infinite, written $m \xrightarrow{\sigma}$.*

2.2.2. Workflow nets

We first define the notion of system which fixes an *initial* marking of the Petri net. In the following, we will mainly be interested in execution sequences starting from this initial marking.

Definition 2.10 (System). *A* system *is a pair (N, m_0) such that:*

- *N is a connected Petri net having at least one place and one transition, and*
- *m_0 is a marking of N called initial marking*

An execution *σ of the system is a finite execution sequence of N starting from m_0 and such that $m_0 \xrightarrow{\sigma} m'$ and no transition of N is enabled in m'.*

We transfer the properties of a Petri net to a system, for example, when we say that a system (N, m_0) is strongly connected, we mean that its underlying Petri net N is strongly connected.

One particular type of system, called *workflow net* (64, 66), is often used to model business processes. A workflow net has a particular *initial* and *final* marking:

Definition 2.11 (Workflow net)**.** *A system (N, m_0) is a* workflow net *if:*

- *There is a single place $i \in P$, called* initial *place, such that $\circ i = \emptyset$.*
- *There is a single place $f \in P$, called* final *place, such that $f \circ = \emptyset$.*
- *Every node $n \in P \cup T$ is on a path from the start place to the final place.*

The initial marking *m_0 of a workflow net is defined as $m_0(i) = 1$ and for any $p \in P$ such that $p \neq i$, $m_0(p) = 0$.*

The final marking *m_f of a workflow net is defined as $m_f(f) = 1$ and for any $p \in P$ such that $p \neq f$, $m_0(p) = 0$.*

A marking reachable from m_0 is said to be a reachable *marking* of *N.*

Let N be the Petri illustrated by Fig. 2.1 without the transition r and its adjacent arcs. The system $(N, [p_1])$ is a workflow net with initial place p_1 and final place p_{14}.

Since all workflow nets have the same initial marking m_0, we often omit to mention m_0, i.e., we talk about a workflow net N instead of a workflow net (N, m_0). We now define how we obtain a connected workflow net by connecting the final place to the initial place of a workflow net.

Definition 2.12 (Connected workflow net)**.** *Let N be a workflow net, the* connected *version $\overline{N} = (\overline{P}, \overline{N}, \overline{F})$* of *$N$ is defined as follows:*

- *$\overline{P} = P$,*
- *$\overline{T} = T \cup \{r\}$, and*
- *$\overline{F} = F \cup \{(f, r), (r, i)\}$.*

where $r \notin T$ is a fresh transition called return *transition.*

The Petri net illustrated by Fig. 2.1 is the connected version of N. By ‘connecting’ a workflow net we turn it into a strongly connected system. This extension of the workflow net is standard in the control-flow analysis literature (64). Connected workflow nets are also often called short-circuit(ed) workflow nets (70, 78).

2.3. Workflow graphs

A workflow graph is a kind of multi-graph with nodes having a *logic type*. More formally:

Definition 2.13 (Workflow graph)**.** *A* workflow graph $W = (N, E, c, l)$ *consists of a multi-graph* $G = (N, E, c)$ *and a mapping* $l : N \to \{\text{TASK}, \text{AND}, \text{XOR}, \text{IOR}\}$ *that associates a* logic *with every node* $n \in N$*, such that:*

1. *The workflow graph has exactly one source and at least one sink.*
2. *For each node* $n \in N$*, there exists a path from the source to one of the sinks that contains* n*.*

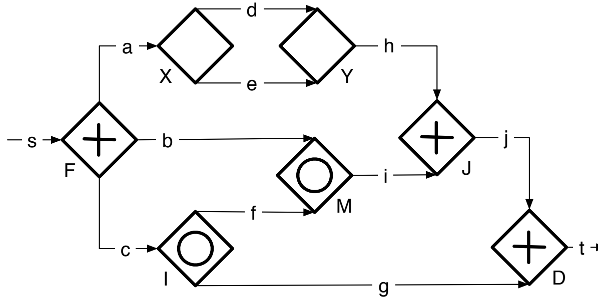


Figure 2.4.: A workflow graph.

Figure 2.4 depicts an acyclic workflow graph. A diamond containing a plus symbol represents a node with AND logic, an empty diamond represents a node with XOR logic, and a diamond with a circle inside represents a node with IOR logic.

A node is a *task*, i.e., a node is of logic type TASK, if and only if it has a single incoming edge and a single outgoing edge. A task is represented by a rectangle. A node with a single incoming edge and multiple outgoing edges is called a *split*. A node with multiple incoming edges and single outgoing edge is called a *join*. For the sake of presentation simplicity, we use workflow graphs composed of only splits and joins. Note that any node with multiple incoming and outgoing edges could be decomposed into a join and a subsequent split.

2.3.1. Acyclic workflow graphs

We often consider workflow graphs that are acyclic in which case the order relation between the elements becomes useful. Let, in the rest of this section, $W = (N, E, c, l)$ be an acyclic workflow graph.

Definition 2.14 (Relationships between elements of an acyclic workflow graph)**.** *Let* $x_1, x_2 \in N \cup E$ *be two distinct elements of* W *such that there is a path from* x_1 *to* x_2*.*

We then say that x_1 precedes x_2*, denoted* $x_1 < x_2$*, and* x_2 follows x_1*.*

Two elements $x_1, x_2 \in N \cup E$ *of* W *are* unrelated*, denoted* $x_1 \mid\mid x_2$*, if* $x_1 \neq x_2$ *and neither* $x_1 < x_2$ *nor* $x_2 < x_1$*.*

Definition 2.15 (Prefix of a workflow graph)**.** *A* prefix *of* W *is a workflow graph* $W' = (N', E', c', l')$ *such that:*

- $N' \subseteq N$*,*
- *for each pair of nodes* $n_1, n_2 \in N$*, if* $n_2 \in N'$ *and* $n_1 < n_2$ *then* $n_1 \in N'$*,*
- *an edge* e *belongs to* E' *if there exists a node* $n \in N'$ *such that* e *is adjacent to* n*,*
- *for each node* $n \in N'$*, we have* $l'(n) = l(n)$*, and*
- *for each edge* $e \in E'$ *such that* $c(e) = (s, t)$*, we have* $c'(e) = (s', t')$*,* $s' = s$*, and* $t' = t$ *if* $t \in N'$ *or* $t' =$ *null otherwise.*

2.3.2. Acyclic workflow graph semantics

As we only use acyclic workflow graphs in this thesis, we define the semantics of acyclic workflow graphs. Note that the acyclic semantics differs from the cyclic semantics only for IOR-joins. We adopt the following IOR-join semantics to simplify the presentation. This IOR semantics, which is explained in detail elsewhere (80), complies with established standards such as BPMN 2.0 (54) and BPEL's (39) *dead path elimination*. We will discuss the semantics of the IOR-join in further details in Chap. 4.

The semantics of workflow graphs is, similarly to Petri nets, defined as a token game. In a workflow graph, the edges carry the tokens (similarly to places in a Petri net) and the nodes represent one or more transitions: If n is a task, executing n removes one token from the single incoming edge of n and adds one token to the single outgoing edge of n. If n has AND logic, executing n removes one token from each of the incoming edges of n and adds one token to each of the outgoing edges of n. If n has XOR logic, executing n removes one token from one of the incoming edges of n and adds one token to one of the outgoing edges of n. If n has IOR logic, n can be executed if and only if at least one of its incoming edges is marked and there is no marked edge that precedes a non-marked incoming edge of n. When n executes, it removes one token from each of its marked incoming edges and adds one token to a non-empty subset of its outgoing edges.

The outgoing edge or set of outgoing edges to which a token is added when executing a node with XOR or IOR logic is non-deterministic, by which we abstract from data-based or event-based decisions in the process. In the following, this semantics is defined formally.

Definition 2.16 (Marking)**.** *A* marking $m : E \to \mathbb{N}$ *of a workflow graph with edges* E *is a multi-set over* E*. When* $m[e] = k$*, we say that the edge* e is marked with k tokens

in m. *When* $m[e] > 0$, *we say that the edge* e is marked *in* m. *The* initial marking m_s *of* W *is such that the source edge is marked with one token in* m_s *and no other edge is marked in* m_s.

Definition 2.17 (Transition). *A tuple* (E_1, n, E_2) *is called a* transition *if* $n \in N$, $E_1 \subseteq \circ n$, *and* $E_2 \subseteq n\circ$.

Definition 2.18 (Acyclic workflow graph semantics). *A transition* (E_1, n, E_2) *is* enabled *in a marking* m *if for each edge* $e \in E_1$ *we have* $m[e] > 0$ *and any of the following propositions holds:*

- $l(n) = \text{TASK}$ *or* $l(n) = \text{AND}$, $E_1 = \circ n$, *and* $E_2 = n\circ$.
- $l(n) = \text{XOR}$, *there exists an edge* e *such that* $E_1 = \{e\}$, *and there exists an edge* e' *such that* $E_2 = \{e'\}$.
- $l(n) = \text{IOR}$, E_1, E_2 *are non-empty,* $E_1 = \{e \in \circ n \mid m(e) > 0\}$, *and, for every edge* $e \in \circ n \setminus E_1$, *there exists no edge* e', *marked in* m, *such that* $e' < e$.

A transition T *can be* executed *in a marking* m *if* T *is enabled in* m. *When* T *is executed in* m, *a marking* m' *results such that* $m' = m - E_1 + E_2$.

Again, the notions of execution sequence, execution, and reachability are defined in a similar way as for Petri nets.

Definition 2.19 (Execution sequence, execution, and reachability). *An* execution sequence *of* W *is an alternate sequence* $\sigma = \langle m_0, T_0, m_1, T_1 ... \rangle$ *of markings* m_i *of* W *and transitions* $T_i = (E_i, n_i, E_i')$ *such that, for each* $i \geq 0$, T_i *is enabled in* m_i *and* m_{i+1} *results from the execution of* T_i *in* m_i.

An execution *of* W *is an execution sequence* $\sigma = \langle m_0, ..., m_n \rangle$ *of* W *such that* $n > 0$, $m_0 = m_s$ *and there is no transition enabled in* m_n.

Let m *be a marking of* W, m *is* reachable from *a marking* m' *of* W *if there exists an execution sequence* $\sigma = \langle m_0, ..., m_n \rangle$ *of* W *such that* $m_0 = m'$ *and* $m = m_n$. *A marking* m *is a* reachable marking *of* W *if* m *is reachable from* m_s.

As the transition between two markings can be easily deduced, we often omit the transitions when representing an execution or an execution sequence, i.e., we write them as sequence of markings.

2.4. Relationship between workflow graphs without IOR and free-choice workflow nets

So far, we defined numerous standard concepts, which we use to model the control-flow of a business process model. In this section, we discuss the relationships between

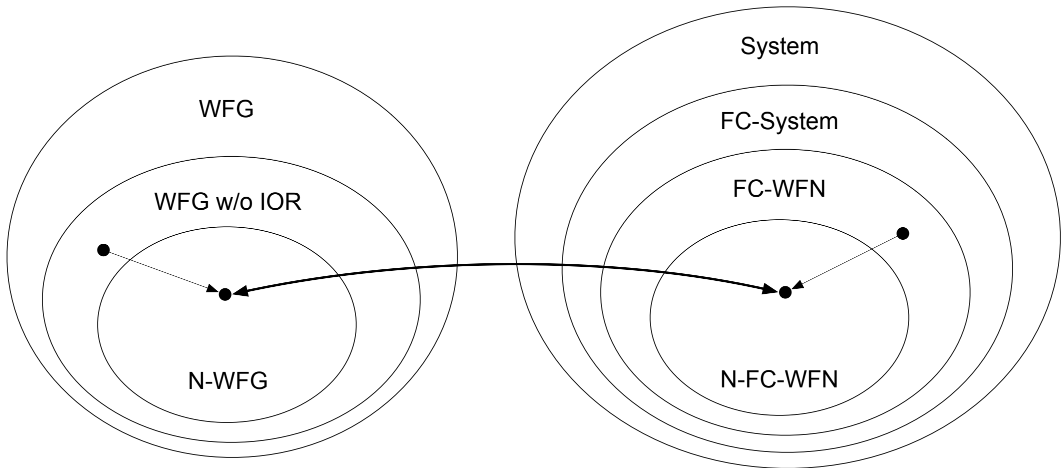


Figure 2.5.: Relation between systems based on workflow graphs (WFG) and free-choice (FC) workflow nets (WFN).

these concepts. In particular, we summarize a result that we describe in more details elsewhere (22), which establishes that workflow graphs without IOR-logic and free-choice workflow nets are isomorphic. We provide a translation from a workflow graph without IOR-join into an equivalent free-choice workflow net and vice versa. This bidirectional translation allows us to use both representations in this thesis and to switch between the two representations.

Fig. 2.5 summarizes the relationships among the different types of systems based on workflow graph and on Petri nets [1]. It also illustrates that a workflow graph without IOR-logic in a normal form (N-WFG) can be translated into an equivalent free-choice workflow-net in normal form (N-FC-WFN). We will define later the normal form of a workflow net, or a workflow graph, and show how to obtain them.

We now provide a translation from a workflow graph into an equivalent free-choice workflow net and vice versa and establish that workflow graphs without IOR-logic and free-choice workflow nets are isomorphic.

Note that we omit tasks from the presentation because they are not relevant to perform a control-flow analysis. The translation between free-choice workflow nets and workflow graphs can also preserve tasks as detailed in the original presentation (22).

The translation is based on the rules shown in Fig. 2.6. The rules match a workflow graph node and its adjacent edges (drawn dashed) with a Petri net pattern and its adjacent arcs (drawn dashed). Note that the dashed arcs/edges are indicative of the context of the rule but are not part of the pattern. Each Petri net pattern on the right hand side is a cluster. The clusters of a Petri net define a partition of the Petri net. To

[1] Although used in the control-flow literature, WFN, i.e., non free-choice WFN, are not depicted in this illustration because they are not used in this thesis.

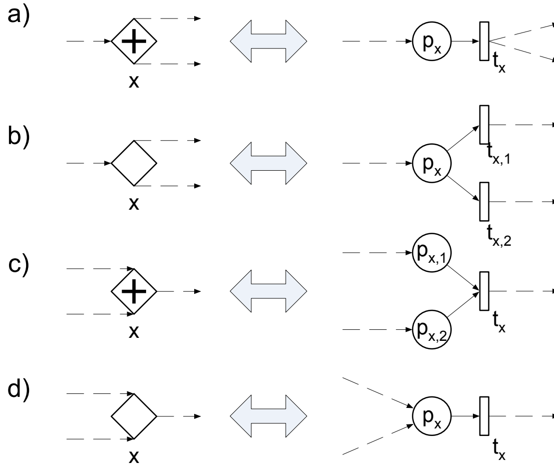


Figure 2.6.: Translating nodes of a workflow graph to corresponding Petri net patterns.

translate from a workflow net to workflow graph, we first partition the workflow net into clusters before matching them. The two directions of the translation are performed in two steps:

1. First, each node (cluster) of the workflow graph (workflow net) is translated to the matching a Petri net pattern (workflow graph node).

2. Then, each edge (arc between clusters, i.e., dashed arc), is translated to an arc (edge).

The rules of Fig. 2.6 only apply if the graph is in a *normal form.*

Definition 2.20 (Normal forms)**.** *We say that a node x of a workflow graph or of a Petri net is* trivial *if it has at most one incoming and at most one outgoing edge; x is said to be* normal *if it has at most one incoming or at most one outgoing edge.*

A workflow graph is normalized *if each node is normal.*

Let N be a Petri net.

A cluster (P', T', F') of N is normalized *if there is at most one node $x \in P' \cup T'$ that is not trivial in N, i.e., x is the only node with more than one incoming or outgoing arc.*

N is normalized *if each node of N is normal and each cluster of N is normalized.*

The original presentation (22) provides a way to obtain the normal form and a formal presentation of the two directions of the translation. For the rest of this thesis we only use workflow graphs and workflow nets in normal form. There is a strong correspondence between workflow graphs and workflow nets in normal form, as sketched in Figure 2.6:

1. The translation rules of Fig. 2.6 define a *bijection* between workflow graphs in normal form (N-WFG) and *corresponding* free-choice workflow nets in normal form (N-FC-WFN).

2. A workflow graph in normal form and its corresponding workflow net in normal form are *isomorphic.*

3. Each workflow graph (workflow net) can be transformed into one in normal form through a sequence of simple semantics-preserving transformation rules.

A proof of this correspondence is provided in the original presentation.

2.5. Control-flow correctness criterions

Now that we have defined our two main control-flow abstractions of a business process model, clarified the relationship between them, and shown the equivalence between free-choice wokflow nets and workflow graphs without IOR-joins, we will define the notions of correctness that we use in this thesis. We will focus on workflow nets for which many correctness notions exists. These notions can be easily transferred to workflow graphs for which we will use only one such notion.

2.5.1. Soundness

The usual notion of correctness for the control-flow of business process models is called *soundness* and was originally defined for workflow nets (66). The soundness of a workflow net ensures that any of its execution can always terminate, that every termination is a 'proper termination', i.e., only the final place is marked, and that every transition can be executed in some execution:

Definition 2.21 (Sound workflow net)**.** *A workflow net $N = (P, T, F)$*[1] *is* sound *if:*

1. *For every reachable marking m of N, there exists an execution sequence σ such that $m \xrightarrow{\sigma} m_f$ and m_f is the final marking of N.*

2. *The final marking of N is the only reachable marking of N which marks the final place.*

3. *For every transition $t \in T$, there is a reachable marking which enables t.*

An violation of the soundness property is indicative of a control-flow error in the underlying business process model. As argued in Chap. 1, a control-flow error in a business process model has severe consequences on its executability.

We will now see that this definition has many equivalent characterizations. We start by introducing additional concepts from Petri net theory:

[1] Remember that since for all workflow nets have the same initial marking m_0, we often omit to mention m_0, i.e., we talk about a workflow net N instead of a workflow net (N, m_0).

Definition 2.22 (Live, bounded, and safe)**.** *Let $N = (P, T, F)$ be a strongly connected Petri net, and let m_0 be the initial marking of the system (N, m_0).*

A transition $t \in T$ is live *if for each reachable marking m of (N, m_0), there exists a marking m' such that m' is reachable from m and t is enabled in m'.*

(N, m_0) is live *if each transition in T is live.*

(N, m_0) is k-bounded *if for each reachable marking m and each place $p \in P$, $m(p) \leq k$.*

(N, m_0) is bounded *if there exist k such that (N, m_0) is k-bounded.*

(N, m_0) is safe *if (N, m_0) is 1-bounded. We say that a marking m of N is a* lack of synchronization *if there is a place $p \in P$ of such that $m(p) > 1$.*

Some markings are undesirable to reach:

Definition 2.23 (Local and global deadlocks)**.** *Let $N = (P, T, F)$ be a workflow net.*

A marking m of N is a local deadlock *if there exist a place $p \in P$ which is not the final place of N, such that m marks p and all the markings reachable from m mark p.*

A marking of N is a global deadlock *if no transition is enabled by m and m marks at least one place different from the final place.*

We say that N contains *a local deadlock (global deadlock) if a reachable marking of N is a local deadlock (global deadlock).*

Consider the reachable marking $m_1 = [p_{10}, p_{12}]$ of the connected workflow net illustrated by Fig. 2.1. This marking is a local deadlock as there is no marking reachable from m_1 which does not mark p_{10} and p_{12} . The marking m_1 is even a global deadlock because no transition is enabled. The marking m_1 being a global deadlock directly implies that the connected workflow net is not live because no transition can get enabled from m_1. Reaching a lack of synchronization, such as $m_2 = [p_{12}, p_{12}]$ for example, indicates that the workflow net is unsafe. a lack of synchronization also indicates that the connected workflow net is unsound because, as we will later, its proper final state ($[p_{14}]$ in our example) cannot be reached from a lack of synchronization (such as m_2 in our example).

Note that a global deadlock is also a local deadlock. A local deadlock m in a connected workflow net N indicates that N is unsound because, by definition of local deadlock, there is a place p that is marked by m and by any marking reachable from m, therefore either the transition in $p\circ$ is never enabled in a marking reachable from m which implies that the connected workflow net is not live, or, in case a reachable marking m' from m enables the transition t in $t\circ$, then m' must be a lack of synchronization because, after executing t, a token must still mark p.

Definition 2.24 (Error marking)**.** *Let N be a workflow net. A reachable marking m of N is an* error marking *if m is a lack of synchronization or m is a global deadlock.*

As we discussed earlier, these notions are closely related:

Theorem 2.25 (Equivalent soundness characterizations)**.** *Let N be a free-choice workflow net.*

1. *N is sound if and only if the connected workflow net $\overline{N}$ is live and bounded. (66)*
2. *N is sound if and only if N is safe and contains no local deadlock. (75)*
3. *N is sound if and only if no reachable marking of N is an error marking. (20)*

The first condition of Thm. 2.25 is leveraged when one uses existing Petri net analysis techniques which establish liveness and boundedness of a strongly connected system.

The second and third conditions allow us to understand the control-flow error as an undesired marking. A lack of synchronization and global deadlock can be recognized by inspection of the marking. A local deadlock m requires knowledge of all the reachable markings form m to be recognized. Therefore, approaches based on state space exploration typically use the third. In the following, we also use the third condition to define soundness of workflow graphs.

2.5.2. Alternative correctness criterions

Requiring soundness is sometimes too restrictive for practical purposes: For some use cases centered around documentation, it might be sufficient that some executions, the so-called 'happy flows', are free of control-flow errors. Moreover, because control-flow analysis techniques abstract from the data constraints, an error marking of a workflow net or a workflow graph might never be reached by any actual execution of the business process model (cf. Sect. 5.3 for more explanations).

To relax the notion of soundness, the two notions of correctness that follow have been proposed in the literature (6, 49). We will come back to these notions in Sect. 5.3 and Sect. 5.4.

Definition 2.26 (Sound execution, weak soundness, and relaxed soundness)**.** *Let N be a workflow net.*

An execution σ is of N sound *if no marking of σ is an error marking.*

N is weakly sound *if there exists a sound execution of N.*

N is relaxed sound *if for each transition $t \in T$, there exists a sound execution executing t.*

While systems specify an initial marking, there are control-flow errors that do not depend on this initial marking. These errors are captured by the notion of *structural soundness*, which we will use in Chap. 3.

Definition 2.27 (Structural soundness)**.** *Let N be a strongly connected Petri net.*

N is structurally live *if there exists an initial marking m_0 such that (N, m_0) is live.*

N is structurally bounded *if for each initial marking m_0, (N, m_0) is bounded.*

N is structurally sound *if N is structurally live and structurally bounded.*

Part II.

Control-Flow Analysis

3

Structural and Dynamic Diagnostic Information

In this chapter, we characterize soundness structurally, i.e., in terms of structural error patterns of the workflow net. We present a polynomial control-flow analysis technique which checks this characterization. Upon detecting a control-flow error, the technique returns two types of diagnostic information: 1. A structural error pattern and 2. a dynamic diagnostic information in the form of a reduced error trace.

3.1. Motivation

As argued in Sect. 1.3, existing approaches present a trade-off between efficiency of the technique and the quality of the diagnostic information that it provides. Regardless of the efficiency requirement, various authors (19, 50) have recognized the need to provide better diagnostic information. Even the error trace, which is currently the best diagnostic information, has some drawbacks: Error traces are often long, and only a portion of the error trace might be relevant to understand the error. Moreover, an error trace exhibits one particular execution of the process leading to an invalid state. The actual 'cause' of an error is an invalid structure in the process.

We adopt the precept that, as the control-flow of business process model is expressed by its structure, a structural error pattern in the business process model is the best representation of the cause of a control-flow error. An error trace gives an dynamic example of an improper execution, which is an useful complement of the error pattern. We aim to combine the consumability of structural diagnostic information and error traces. We also require the analysis technique to be efficient, i.e., to have a polynomial worst case time complexity.

In this chapter, we present analysis technique providing two types of diagnostic in-

formation: a structural error patterns and a reduced error trace. The technique has a polynomial time worst case complexity: In Sect. 3.2, we present the structural error patterns that we search for, we introduce the structural soundness characterization that we obtain, and we give an intuitive explanation of why these structural patterns are indicative of a control-flow error. In Sect. 3.3, we prove that the structural error patterns are indicative of control-flow error. Moreover, the proofs provide a procedure to derive an error trace in quadratic time based on a structural error pattern. In Sect. 3.4, we prove both directions of the structural characterization of soundness. In Sect. 3.5, we present a technique to detect structural error patterns in polynomial time and prove its correctness.

3.2. Structural errors

In this section, we present the control-flow patterns that we use to identify control-flow errors.

3.2.1. An existing characterization

We start with the concepts of *circuits*, *handles*, and *bridges* (see Fig. 3.1). These concepts are at the root of an existing (14) characterization of structural soundness, which will help us in our reasoning.

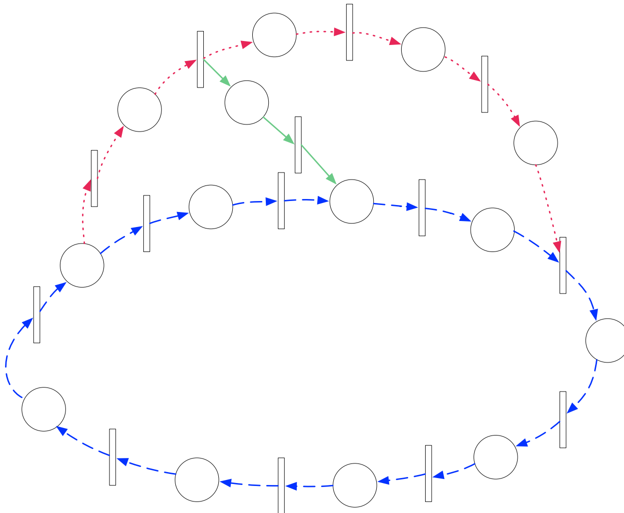


Figure 3.1.: A circuit (dashed line), a handle (dotted line), and a bridge (plain line).

Let, for the rest of this section, $N' = (P', T', F')$ and $N'' = (P'', T'', F'')$ be two subnets of a Petri net $N = (P, T, F)$.

Definition 3.1 (Circuits, handles, and bridges). *A* circuit Ω *in* N *is a path in* N *from a node to itself such that all other elements of* Ω *are pairwise distinct.*

A handle H *of* N' *is a non-trivial path in* N *from a node* a *of* N' *to a node* b *of* N' *such that* H *is disjoint from* N' *aside from* a *and* b[1]. *We say that* H *is a* P/T-handle *if* $a \in P'$ *and* $b \in T'$, *and that* H *is a* T/P-handle *if* $a \in T'$ *and* $b \in P'$.

A bridge B *between* N' *and* N'' *is a non-trivial path in* N *from a node* a *in* N' *to a node* b *in* N'' *such that* B *is disjoint from the elements of* N' *aside from* a *and is disjoint from* N'' *aside from* b. *The bridge* B *is a* T/P-bridge *if* $a \in T'$ *and* $b \in P''$.

Consider the connected workflow net illustrated by Fig. 3.2, the path $H = \langle p_1, t_1, p_2, t_3, p_6, t_7\rangle$ is a P/T-handle of the circuit $\Omega = \langle p_1, t_2, p_5, t_6, p_7, t_7, p_8, r, p_1\rangle$ and the path $B = \langle t_1, p_3, t_4, p_7\rangle$ is a T/P-bridge between H and Ω.

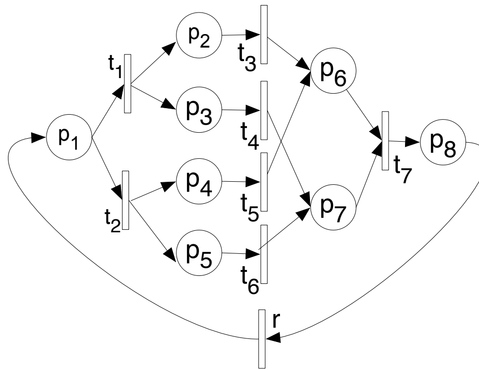


Figure 3.2.: A sound connected workflow net containing interesting circuits, handles, and bridges.

These concepts allowed Esparza and Silva (14) to characterize structurally live and structurally bounded free-choice Petri nets, i.e., structurally sound (cf. Def. 2.27) free-choice Petri nets, in terms of circuits, handles, and bridges as follows:

Theorem 3.2 (Structural soundness (14)). *Let* $N = (P, T, F)$ *be a free-choice Petri net.* N *is structurally sound if and only if:*

1. N *is strongly connected.*
2. *No circuit of* N *has a T/P handle.*
3. *Every P/T handle of a circuit is bridged to it through a T/P bridge.*

Intuitively, a circuit with a T/P handle hints at a lack of synchronization and a circuit with a P/T handle hints at a deadlock: Executing the transition at the beginning of the

[1] The arcs of the subnet and the handle have to be considered: If we consider an path with only two elements a and b, the arc between the two elements being outside or inside of the subgraph makes the difference between an handle and a path in the subgraph.

T/P handle results in two tokens: one on the circuit and one on the handle. Following either the circuit or the handle, both tokens can move on their respective paths towards the last place of the handle, which is also on the circuit. When both tokens reach this place, there is a lack of synchronization. On the opposite, if a token reaching the first place of a P/T handle 'follows' the P/T handle, it 'removes' a token from the circuit. In the absence of T/P bridge no token will be 'returned' to the circuit and upon reaching the transition at the intersection of the handle and the circuit, the token is blocked because there is no token on the circuit that can enable this transition.

A connected workflow net is a special type of strongly connected free-choice Petri net. Finding a circuit with a T/P-handle or a circuit with a P/T-handle without a T/P-bridge would allows us to identify a violation of structural soundness according to Thm. 3.2. A naive approach to check for these patterns would require to check, for every circuit of the graph for either of the two type of handle. Unfortunately, this approach would be inefficient because the number of circuits in a graph is potentially exponential with respect to the size of the graph. There is no approach in the literature which would allow us to check Thm. 3.2 in polynomial-time. Moreover, being able to check this characterization would ensure that the connected workflow net is structurally sound, i.e., that it is bounded and that there exists some arbitrary initial marking of the underlying net that is live. To ensure the soundness of a workflow net, we would, in addition to structural soundness, want to know whether the initial marking is live.

In the following, we will use the concept of a siphon to build our structural characterization.

3.2.2. Siphons

As defined by Def. 2.7, a siphon is a set S of places of a Petri net such that $\circ S \subseteq S\circ$. The set of places $S = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_{10}, p_{13}, p_{14}\}$, highlighted in Fig. 2.1, is an example of a siphon. By definition, any transition allowing to bring a token in a siphon also removes at least a token of the siphon. It implies that, once empty, a siphon can never get marked (again):

Proposition 3.3 (Unmarked siphons remain unmarked (9))**.** *Let N be a Petri net. Let S be a siphon of N.*

If a marking m of N does not mark S, there exists no marking m', reachable from m, which marks S.

By Proposition 3.3, all transitions in the subnet generated by an unmarked siphon are dead[1], i.e, not live.

A particularly interesting type of siphon are the ones we call decreasing quasi-component siphons:

[1] This property has lead to calling siphons 'deadlock'. We avoid this naming to avoid confusion with a deadlock marking.

Definition 3.4 (DQ siphon)**.** *Let N be a Petri net and S be a siphon of N. S is a* decreasing quasi-component siphon, DQ siphon for short, *iff for any transition t in N, we have $|t \circ \cap S| \leq 1$.*

The key property of a DQ siphon is that there is no transition which has more than one post place within the siphon. In combination with the general property of a siphon that any transition allowing to bring a token in a siphon also removes at least a token of the siphon, we have that the marking of a DQ siphon is monotonously decreasing during any execution:

Proposition 3.5 (The marking of DQ siphons is monotonously decreasing)**.** *Let N be a Petri net, S be a DQ siphon of N, m_1 be a marking of N.*
For any execution sequence σ of N such that $m_1 \xrightarrow{\sigma} m_2$, we have $m_1(S) \geq m_2(S)$.

Proof. Can be proven by induction on the length of σ using the property of DQ-siphon that, for any transition t in N, we have $|t \circ \cap S| \leq 1$ (cf. Def. 3.4) and thus executing any transition either maintains or decrease the number of tokens in S. □

By definition, a (DQ) siphon is a set of places. By abuse of language, in the context of graph structures, we often refer to the subnet generated by a given siphon S simply as the siphon S.

3.2.3. Error patterns

We propose an alternative characterization of soundness for workflow nets that is completely based on structural error patterns:

Theorem 3.6 (Structural characterization of soundness)**.** *Let N be a free-choice workflow net.*
N is unsound iff at least one of the following statements holds:

1. *There exists a siphon that does not contain the initial place.*
2. *There exists a DQ siphon with a P/T handle.*[1]
3. *There exists a simple path from some element to the final place with a T/P handle.*

(The proof is deferred to Sect. 3.4)

The first condition was already used (e.g. Kemper and Bause (41)) to establish the soundness of a structurally sound connected workflow net. The second and third conditions are novel but, as we will prove later, can be related to the notions presented by Esparza and Silva of circuits with a T/P handle or circuits with a P/T handle without a T/P bridge (cf. Thm. 3.2).

[1] Strictly speaking this condition should be written: There exists a DQ siphon generating a subgraph which has a P/T handle.

We will present a proof of Thm. 3.6 in Sect. 3.4. In the remainder of this section we present an example of each error pattern in this structural characterization and explain informally why each pattern of the characterization is an indication of unsoundness.

3.2.3.1. A siphon that does not contain the initial place

In a workflow net, a siphon that does not contain the initial place, i.e., that is not marked by the initial marking, is clearly undesirable because, by Proposition 3.3, the places of the siphon are not marked by any execution, i.e., the area is 'dead'. This implies that the transitions in the subgraph generated by the siphon are never enabled which violates the third condition of the classical definition of soundness (Def. 2.21). An example of such a siphon is highlighted in Fig. 3.3.

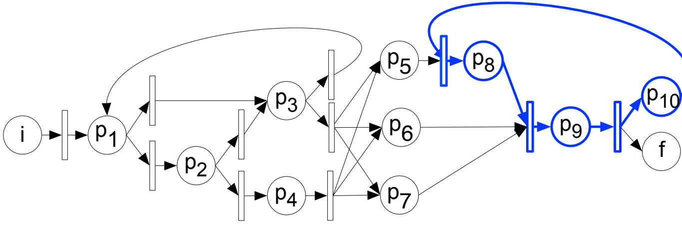


Figure 3.3.: A workflow net with a siphon that does not contain the initial place.

In Fig. 3.3, we can observe that the highlighted siphon cannot get marked by any execution: the area defined by the siphon is 'dead'. Moreover, it is clear that any reachable marking which marks a place in $\{p_5, p_6, p_7\}$, is a local deadlock. In this particular example, every execution eventually marks these three places and thus every execution contains a local deadlock, i.e., no execution terminates properly.

3.2.3.2. There exists a DQ siphon with a P/T handle

Fig. 3.4 illustrates a workflow net whose connected version has been discussed already in Sect. 2.2. The subnet generated by the DQ siphon $S = \{i, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_{10}, p_{12}, f\}$ (highlighted elements connected by plain arrows) has a P/T handle $H = \{p_7, t_{13}, p_9, t_{15}\}$ (highlighted elements connected by dashed arrows).

As argued in Sect. 3.2.3.1, a siphon that is not marked is undesirable. Therefore, no siphon may be empty initially and we must not be able to empty siphon. We will now describe how a P/T handle of a DQ siphon gives us a strategy to 'empty' the siphon from its token.

Let S be a DQ siphon of a workflow net. If there is a place $p \in S$ such that $p\circ \not\subseteq \circ S$, executing a transition t in $p\circ \setminus \circ S$ would remove at least one token from S because, if any place in $t\circ$ was part of S, then t would belong to $\circ S$. Note that the first place of P/T handle is always such a place otherwise the first transition of the handle would belong

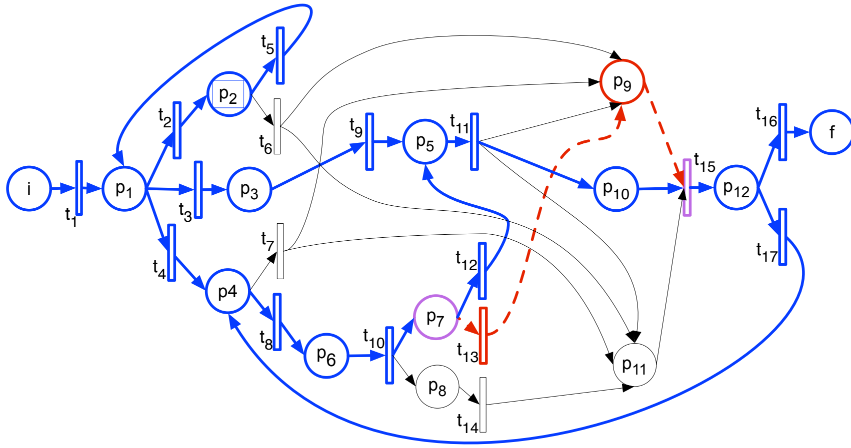


Figure 3.4.: A DQ siphon with a P/T handle.

to the subnet generated by S which would contradict the disjointness assumption of the handle.

Assuming that S contains the initial place, the initial marking marks one place in S. As S is DQ siphon, the number of tokens marking S cannot increase. Therefore, to empty the siphon, it is enough for an execution to reach a marking which marks the first place of the handle and then to execute the first transition of the handle.

As argued earlier, once S is empty, there is no reachable marking enabling a transition of the subnet generated by S. Therefore, we can reach a marking which is a local deadlock by moving the token along the handle until reaching its last place where the token will be stuck.

In the example of the workflow net of Fig. 3.4, the execution sequence $\langle[i], [p_1], [p_4], [p_6], [p_7, p_8], [p_9, p_8]\rangle$ leads to emptying S by following the P/T handle when reaching p_7 resulting in the marking $[p_9, p_8]$ which is a local deadlock because the place p_{10} will never get a token, therefore the transition t_{15} will never be enabled and the token on p_9 is stuck forever.

3.2.3.3. There exists a simple path to the final place with a T/P handle

A simple path to the final place is, unlike the previous two patterns, an indication that a lack of synchronization (cf. Def. 2.22) might[1] be reachable.

Fig. 3.5 illustrates a workflow where a simple path (connected by highlighted plain arrows) to the final place and its T/P handle (connected by dashed arrows) are highlighted.

The execution sequence $\langle [i], [p_1], [p_4], [p_6], [p_7, p_8], [p_5, p_8], [p_{11}, p_8], [p_{11}, p_{11}] \rangle$ ends in a lack of synchronization.

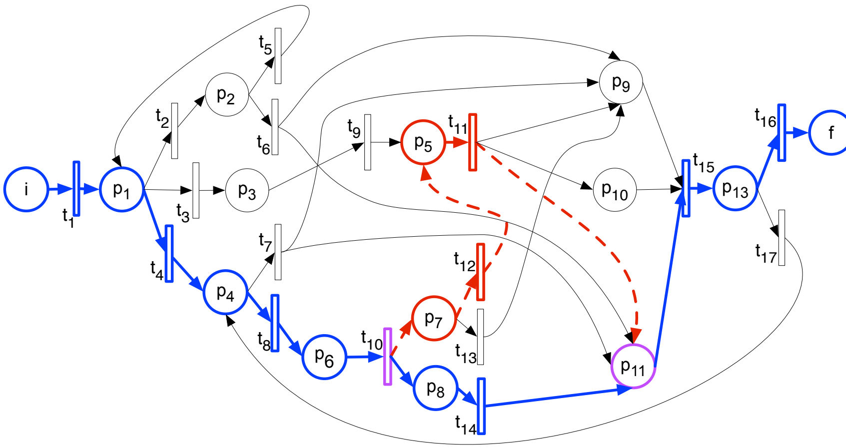


Figure 3.5.: A simple path to f with a T/P handle.

One strategy to obtain a lack of synchronization of this error pattern is to build an execution where the first transition t of the handle is executed. The marking resulting from the execution of t marks the first place on the handle and the place on the path following t. We will show later that the two tokens, marking these places, can be moved on the handle or on path with a certain independence: they can move without requiring to consume the other token. Thus, the token on the handle can be moved until reaching the last place of the handle. At this point there are two tokens on the path to the final place. Again these two tokens can be moved independently until both are on the end up on the same place. This can occur on the final place or earlier.

[1]Due to another error pattern, the execution might end in a global deadlock and prevent the lack of synchronization marking to be reached.

3.3. Correspondence of the error patterns to an error trace

In this section, we prove that we can obtain, in polynomial time, an error trace out of a siphon that does not contain the initial place, a DQ siphon with a P/T handle, or a simple path to the final place with a T/P handle.

We first define a few important Petri net concepts that will help us in building an error trace. Then we will show that we are able to build a reduced error trace for each error pattern in polynomial time.

3.3.1. Basic concepts to build an error trace in quadratic time

We start with the notion of allocation which allows us to specify an 'execution strategy':

Definition 3.7 (Allocation (9)). *Let $N = (P, T, F)$ be a Petri net and C be a subset of all the clusters of N.*[1]

An allocation of C *is a function $\alpha : C \to T$ such that $\alpha(c) \in c$ for every $c \in C$.*

A transition t is allocated by α *if $t = \alpha([t])$.*

If C contains all the clusters of N, α is called a total allocation *of N.*

An execution (or an execution sequence) σ agrees with *an allocation α if each transition $t \in \sigma$ is allocated by α.*

A path π agrees with *an allocation α if each transition $t \in \pi$ is allocated by α*

An allocation α of C points to *a set of clusters C' if for every place $p \in P$ there exists a path π from p to a place in C' such that all the transitions of π are allocated by α.*

The notion that an allocation points to a cluster, or a set of clusters, indicates that the tokens are 'driven' to the cluster by executions following the allocation.

The following lemma establishes that an allocation pointing to a set of clusters always exists for strongly connected free-choice Petri nets. This is a known result which is proven in the literature part of the 'Pointing allocation lemma' (9).

Lemma 3.8 (Existence of an allocation (9)). *Let C be a non-empty set of clusters of a strongly connected free-choice Petri net $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ and $\overline{C}$ be the set of clusters of $\overline{N}$ that do not belong to C.*

There exists an allocation α with domain $\overline{C}$ that points to C.

We can now show the following lemma which we use to build execution sequences following a given allocation in quadratic time.

Lemma 3.9 (An execution sequence to mark a place). *Let $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ be a free-choice workflow net, p_1 and p_2 be two places in $\overline{P}$, and α be a total allocation of N pointing to $[p_2]$.*

[1] The usual definition of allocation requires that every cluster of C contains at least one transition. This is always the case in our setup because we work with strongly connected Petri nets.

If a marking m of N marks p_1, we can obtain, in quadratic time, an execution sequence σ such that $m \xrightarrow{\sigma} m'$ and at least one of the following propositions is true:

1. *m' marks p_2,*
2. *m' marks the final place f of N, or*
3. *m' is an error marking.*

Proof. Let $C = \{[p_2]\}$ and $\overline{C}$ be the set of clusters in $\overline{N}$ that are not contained in C. By Lemma 3.8, there exists an allocation α of $\overline{C}$ pointing to C.

We will use α to 'drive' the construction of σ in quadratic time. To do so we start with some necessary definition and the proof of some properties:

Let $R = (P', T', F')$ be a subnet of $\overline{N}$ such that:

- $p \in P'$ iff there exists a path in $\overline{N}$ from a place marked by m to p that contains only transitions allocated by α,
- $t \in T'$ iff t is allocated by α, and
- $F' = F \cap ((P' \times T') \cup (T' \times P'))$.

It can be easily proven that any execution sequence σ_α agreeing with α can mark only places in P' by induction on the length of σ_α.

Let S be a set of places and m^* be a marking. We call *marked border of m^* with respect to S*, the set of places B such that for each place $p \in B$, there exists a path π^* from p to any place $p' \in S$ such that π^* contains no place, other than p and possibly p', that is marked by m^* .

Let $S = \{p_2\}$ and B be the marked border of m with respect to S. Figure 3.6 illustrates schematically R, S, and B. The highlighted path represents π. The dashed lines represent portions of $\overline{N}$, which do not belong to R. The transitions and their connection to $\overline{N}$ are omitted from the illustration. When a place contains a dot (*token*), it means that the place is marked.

Let σ_1 be an execution sequence obtained by executing any allocated and enabled transitions in $B\circ$ until no allocated transition in $B\circ$ is enabled or either p_2 or f is marked. Note that by the free-choice property, if any transition in a cluster c is enabled, then $\alpha(c)$ is also enabled. Also note that B and thus $B\circ$ are updated at every transition execution. Figure 3.7 illustrates the result of executing σ_1.

We define the function $\Delta(m^*, S, R) = k$ such that k is the number of places that are on a path in R from a place in the marked border of m^* with respect to S, to a place in S.

We show that for each transition $t \in \sigma_1$ such that t is enabled by m, when t is executed, the marking m' resulting from the execution of t has the property $\Delta(m', S, R) < \Delta(m, S, R)$: By definition of σ_1, $\alpha([t]) = t$. By definition of R, for each place $p \in \circ t$, R

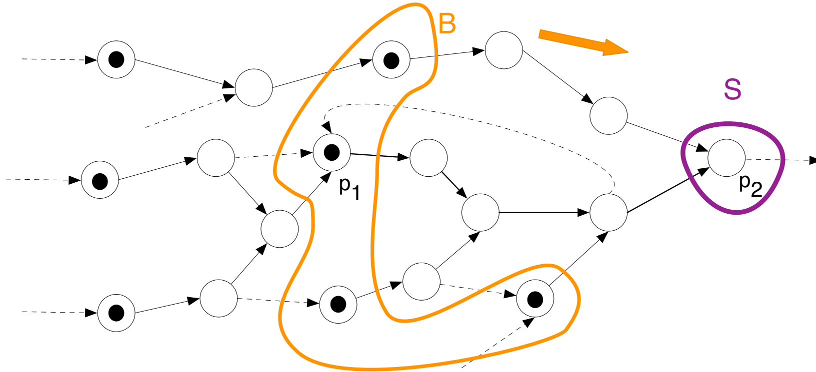


Figure 3.6.: Illustration of S and its marked border B.

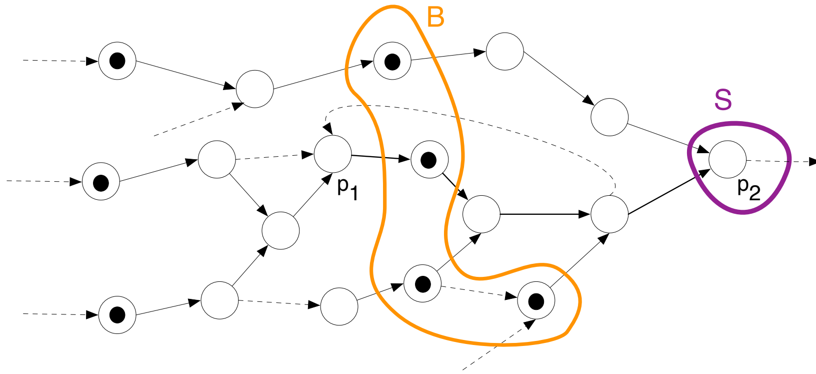


Figure 3.7.: Illustration of S and its marked border B after σ_1.

contains only one transition in $p\circ$: the transition t. Therefore, we have $\Delta(m', S, R) = \Delta(m, S, R) - |\circ t|$.

As $\Delta(m, S, R) \leq |P|$ and Δ is strictly decreasing with any transition of σ_1, we know that the number of transitions in σ_1 is bounded by $|P|$.

Let m_1 be the marking such that $m \xrightarrow{\sigma_1} m_1$. If p_2 or f is marked by m_1, we are done: we constructed $\sigma = \sigma_1$ in quadratic time. In the following we only deal with the case where m_1 does not mark p_2 or f and there is no transition in $B\circ$ enabled by m_1.

We now show that for any place $p \in B$ and each marking m_i reachable from m_1, $m_i(p) \geq 1$. One place $p' \in \circ\alpha([p])$ is not marked by m_1. We show that p' never gets marked and thus $\alpha([p])$ is never enabled and $m_i(p) \geq 1$. We proceed by contradiction:

Suppose that there exists a marking m^* and an execution sequence σ^* such that $m_1 \xrightarrow{\sigma^*} m^*$ and $m^*(p') > 0$. We can assume without loss of generality that p' is the place in $\circ(B\circ)$ which is not marked by m_1 and is first marked during σ^*. As $m_1 \xrightarrow{\sigma^*} m^*$ and $m^*(p') > 0$, there exists a path π in R from a place p^* such that $m_1(p^*) > 0$ to p' such that no place on π, other than p^*, is marked by m_1. Thus, p^* belongs to B and

therefore no transition in $p^*\circ$ is enabled by m_1. To enable $\alpha([p^*])$, another place than p' in $\circ\alpha([p])$ which is not marked by m_1 would have needed to be marked by σ^* which contradicts our supposition that p' is the first.

We have established that there is, and there will be, no enabled transition in any cluster of B. If no transition at all is enabled by m_1, we are done: we constructed $\sigma = \sigma_1$ in quadratic time and m_1 is a global deadlock. In the following we only deal with the case where at least one transition is enabled by m_1 and hence, by construction of R, one place preceding B in R is marked.

Now that we have proven the main properties of σ_1, we show how we can iteratively assign S and continue to build an execution until finding an error. Let B become S_2 and a B^* be a new set of place initially equal to B.

Move iteration: Let B_2 be the marked border of m_1 with respect to S_2. Figure 3.8 illustrates S_2 and its marked border B_2. Let σ_2 be the execution sequence obtained

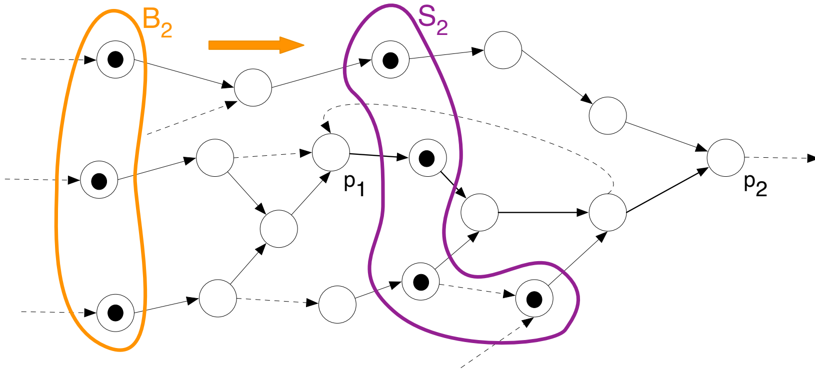


Figure 3.8.: Illustration of S_2 and its marked border B_2.

by executing any allocated and enabled transitions in $B_2\circ$ until no transition in $B_2\circ$ is enabled, and let m_2 be the marking such that $m_1 \xrightarrow{\sigma_2} m_2$. As argued earlier for σ_1, the number of steps in σ_2 is bounded by $|P|$. If $B_2 \cap S \neq \emptyset$, then a place in B_2 carries two tokens and m_2 is a lack of synchronization. If not, then either m_2 is a global deadlock, or there exists a transition preceding B_2 which is enabled in m_2. In the latter case, we repeat the move iteration with $S_3 := B_2$ and $B^* := B^* \cup B_2$.

As the number of places is finite and the number of places on a path to a place in B^* is strictly decreasing at each move iteration, the number of times the step is repeated is bounded by $|P|$. As each step has a maximum of $|P|$ transitions and there is a maximum of $|P|$ steps, the length of σ is bounded by $|P|^2$.

□

The following lemma allows us to obtain, for a given path, a total allocation such that the path agrees with the allocation and the allocation points to the last element of the path.

Lemma 3.10 (Path allocation)**.** *Let N be a strongly connected free-choice Petri net.*

For any simple path π in N, there exists a total allocation α of N such that α points to the last element of π and for every transition in π we have $\alpha[t] = t$.

Proof. Let C be the set of clusters containing the elements of $\pi \setminus \{p_2\}$. Let $\overline{C}$ be the set of clusters in $\overline{N}$ that are not contained in C. By Lemma 3.8, there exists an allocation α_1 of $\overline{C}$ pointing to C.

We first show a that, for each cluster c in C, there is one and only one transition in c which is in π. This will help us in building an allocation α_2 of the clusters in C:

For each cluster c in C, we have: $|T \cap c \cap \pi| = 1$ because

1. There exists at least one transition $t \in c$ such that $t \in \pi$ because, by definition, c contains at least one element x of c if x is a transition, then $t = x$. If x is a place, by definition of a path $x \circ \cap \pi \neq \emptyset$ (remember that we do not consider $[p_2]$) and by definition of cluster $x\circ \subset [x]$.

2. There can be only one transition $t \in c$ such that $t \in \pi$: Suppose that two distinct transitions t_1 and t_2 both belong to c and π. By cluster definition, $\circ t_1 \cap \circ t_2 \neq \emptyset$. By free-choice property, they have $\circ t_1 = \circ t_2$. Both sets $\circ t_1$ and $\circ t_2$ are equal and contain more than one place because t_1 and t_2 are on a simple path and thus they have a distinct place preceding them on π. Thus t_1, t_2 and $\circ t_1$ form an extended free-choice pattern which we ruled out in our definition of free-choice Petri net.

As we have $|T \cap c \cap \pi| = 1$ for each cluster c in C, we can define the allocation α_2 of the clusters in C, for any transition t in a cluster of C we have $\alpha_2[t] = (\circ t) \circ \cap \pi$ and for every place $p \in C$ we have $\alpha_2([p]) = p \circ \cap \pi$. By definition of π and of pointing allocation, α_2 points to p_1.

□

Lemma 3.9 has the two following corollaries for workflow nets. These corollaries will allow us to simplify some proofs in the rest of this section.

Lemma 3.11 (Alternative error marking)**.** *Let $N = (P, T, F)$ be a free-choice workflow net and m be a reachable marking of N such that m marks two places on a simple path π to the final place of N.*

Then, an execution sequence σ, such that $m \xrightarrow{\sigma} m'$ and m' is an error marking, can be obtained in quadratic time.

Proof.

Let $\overline{N}$ be the connected version of N and f be the final place of $\overline{N}$.

We first show that there exists a total allocation α of $\overline{N}$ pointing to f. This allocation will 'drive' the construction of σ.

Let C be the set of clusters containing the elements of $\pi \setminus \{f\}$. Let $\overline{C}$ be the set of clusters in $\overline{N}$ that are not contained in C. By Lemma 3.8, there exists an allocation α_1 of $\overline{C}$ pointing to C.

We first show a property which will help us in building an allocationα_2 of the clusters in C, for each cluster c in C, we have: $|T \cap c \cap \pi| = 1$ because

1. There exists at least one transition $t \in c$ such that $t \in \pi$ because, by definition, c contains at least one element x of c if x is a transition, then $t = x$. If x is a place, by definition of a path $x \circ \cap \pi \neq \emptyset$ (remember that we do not consider $[f]$) and by definition of cluster $x \circ \subset [x]$.

2. There can be only one transition $t \in c$ such that $t \in \pi$: Suppose that two distinct transitions t_1 and t_2 both belong to c and π. By cluster definition, $\circ t_1 \cap \circ t_2 \neq \emptyset$. By free-choice property, they have $\circ t_1 = \circ t_2$. Both sets $\circ t_1$ and $\circ t_2$ are equal and contain more than one place because t_1 and t_2 are on a simple path and thus they have a distinct place preceding them on π. Thus t_1, t_2 and $\circ t_1$ form an extended free-choice pattern which we ruled out in our definition of free-choice Petri net.

Thanks to this property, we can define the allocation α_2 of the clusters in C, for any transition $t \in C$ we have $\alpha_2[t] = (\circ t) \circ \cap \pi$ and for every place $p \in C$ we have $\alpha_2([p]) = p \circ \cap \pi$. By definition of π and of pointing allocation, α_2 points to p_1.

By Lemma 3.9 we can build, in quadratic time, an execution sequence σ (agreeing α) of N that $m \xrightarrow{\sigma} m'$ and m' is an error marking (in which case we are done) or m' marks f. In the following, we will consider the latter case.

It can be proven by an induction on the length of σ that, for any marking m^* reachable from m by any execution sequence σ^* agreeing with α, $m^*(\pi) \geq 2$. The main argument for the induction step is that, because π is a simple path and $\overline{N}$ is free-choice, executing any transition agreeing with α cannot consume both tokens.

We know that a place p^*, preceding f on π, is marked by m' because m' is not an error marking and $m'(\pi) \geq 2$. The final place f remains marked by any execution sequence agreeing with *alpha* because no transition of $[f]$ is allocated by α. By Lemma 3.9, we can continue to build σ in quadratic until reaching a marking m'' such that either is an error marking or $m''(f) > 2$ which is a lack of synchronization, i.e., an error marking.

□

The second corollary is well known in the literature: it is a direct violation of one of the clauses of the classical definition of soundness (See Def. 2.21).

Corollary 3.12 (Improper termination)**.** *Let $N = (P, T, F)$ be a free-choice workflow net and m be a reachable marking of N such that m marks the final place of N and another place.*

Then, an execution sequence σ, such that $m \xrightarrow{\sigma} m'$ and m' is an error marking, can be obtained in quadratic time.

Proof. This is a special case of Lemma 3.11 where the 'second' token on π is already marking f. □

Finally, we prove one more lemma allowing us to obtain for a workflow net, in quadratic time, an execution moving a token along a path until reaching the last place of the path or reaching an error marking.

Lemma 3.13 (Execution sequence following a path). *Let $N = (P, T, F)$ be a free-choice workflow net, p_1 and p_2 be two places in P, and π be a simple path from p_1 to p_2 in N.*

If a marking m of N marks p_1, an execution sequence σ, such that $m \xrightarrow{\sigma} m'$ and m' marks p_2 or m' is an error marking:

Proof. Let $\overline{N}$ be the connected version of N. By Lemma 3.10, there exists a total allocation α of $\overline{N}$ allocating the transitions of π and pointing to p_2.

By Lemma 3.9, we can obtain in quadratic time an execution sequence σ agreeing with α such that $m \xrightarrow{\sigma} m'$ and m' marks p_2, m' is an error marking, or m' marks f. If m' marks p_2 or m' is an error marking, we are done. We consider the case where none of these two conditions is satisfied and thus m' marks f. As σ agrees with α and α allocates the transitions of π, p_2 would have needed to be marked during for m' to not mark a place of π, therefore m' must mark a place of π because. By 3.12, we can compute in quadratic time an execution sequence σ' such that $m' \xrightarrow{\sigma'} m''$ and m'' is an error marking. ☐

3.3.2. Error trace corresponding to a siphon that does not contain the initial place

A siphon that does not contain the initial place is not marked by any execution. In fact, any execution where we try to mark a place in the siphon will end in an error marking:

Theorem 3.14 (A non-marked siphon can be used to build an error trace in quadratic time). *Let $N = (P, T, F)$ be a free-choice workflow net, S be a siphon of N, and m be a reachable of N.*

An execution σ, such that $m \xrightarrow{\sigma} m'$ and m is an error marking, can be computed in quadratic time.

Proof. Let $p \in S$. By the workflow net definition there exists a simple path from the initial place of N. By Lemma 3.13 we can compute in quadratic time an execution trace σ such that $m \xrightarrow{\sigma} m'$ and m' either marks p or is an error marking. By the empty siphon property, m' cannot mark p. Therefore m' is an error marking. ☐

3.3.3. Error trace corresponding to DQ siphon with a P/T handle

A siphon with a P/T handle also allows us to compute an error trace in quadratic time:

Theorem 3.15 (A siphon with a P/T handle can be used to build an error trace in quadratic time)**.** *Let $N = (P, T, F)$ be a free-choice workflow net, S be a siphon of N with a P/T-handle H.*

An execution σ, such that $m_0 \xrightarrow{\sigma} m$ and m is an error marking, can be computed in quadratic time.

Proof. Let p_1 be the first place of the handle. By definition of workflow net, there is a simple path from the initial place of N to p_1. By Lemma 3.13, we can obtain an execution sequence σ_1 such that $m_0 \xrightarrow{\sigma_1} m_1$ and either m_1 is an error marking in which case $\sigma = \sigma_1$ or m_1 marks p_1. By the DQ siphon property (cf. Proposition 3.5), the number of tokens in S cannot increase. The number of tokens is originally 1. As the last transition of σ_1 removes one token from S, $m_1(S) = 0$. By Thm. 3.14, we can compute an error trace in quadratic time. □

3.3.4. Error trace corresponding to a simple path to the final place with a T/P handle

We now show how we can build an error trace when there exists a simple path from an element of N to the final place of N with a T/P handle:

Theorem 3.16 (A simple path to the final place with a T/P handle can be used to build an error trace in quadratic time)**.** *Let $N = (P, T, F)$ be a free-choice workflow net, Ω be a circuit of N and H be a T/P-handle of c.*

An execution σ, such that $m_0 \xrightarrow{\sigma} m$ and m is an error marking, can be computed in quadratic time.

Proof. Let α be an allocation pointing to a place p_1 preceding the first transition t of H. By Lemma 3.13, we can obtain an execution sequence σ_1 such that $m_0 \xrightarrow{\sigma_1} m_1$ and either m_1 is an error marking in which case $\sigma = \sigma_1$ or m_1 marks p_1. Let p be the last node of H. As H is a T/P-handle of Ω, t and p are on Ω, t is a transition, and p is a place. Let π be part of Ω from t to p and β be a total allocation pointing to p, which allocates the transitions in H and π. We show that such allocation exists by showing that for each cluster c containing an element of H or π we have $|H \cap c| = 1$ or $|\pi \cap c| = 1$, and there exist no pair of transitions $t_1 \in \pi$ and $t_2 \in H$ such that t_1 and t_2 belong to the same cluster.

1. There exists at least one transition $t \in c$ such that $t \in \pi$ ($t \in H$) because by definition c contains at least an element x of c if x is a transition, then $t = x$. If x is a place, by definition of path (handle), $x \circ \cap \pi \neq \emptyset$ ($x \circ \cap H \neq \emptyset$) and by definition of cluster $x\circ \subset [x]$.

2. There can be only one transition $t \in c$ such that $t \in \pi \cup H$: Suppose that two distinct transitions t_1 and t_2 that both belong to c and $\pi \cup H$. By cluster definition $\circ t_1 \cap \circ t_2 \neq \emptyset$. By free-choice property they have $\circ t_1 = \circ t_2$. Both sets $\circ t_1$ and $\circ t_2$

are equal and contain more than one place because t_1 and t_2 are on (disjoints) simple paths and thus they have a distinct place preceding them on π and H. Thus t_1, t_2 and $\circ t_1$ form an extended free-choice pattern which we ruled out in our definition of free-choice Petri net.

It is clear that during any execution sequence σ_2 following β such that $m_1 \stackrel{\sigma_2}{\to} m_2$ and no marking before m_2 marks p, there is always a token on H and a token on π. By Lemma 3.9, we can compute in quadratic time execution sequence σ_2 such that $m_1 \stackrel{\sigma_2}{\to} m_2$ and m_2 is either an error marking, m_2 marks f, or $m_2(p) > 1$. In case m_2 is an error marking, we are done and $\sigma = \sigma_1\sigma_2$. We now consider the two other cases. As the marking m_2' preceding m_2 in σ_2 marks a place on π and a place on H, m_2 still marks a place on Ω or a place on π because the last transition on π is distinct from the last transition on H. Therefore m_2 marks two places and there exists a simple path from one of the marked place to the final place which contains the other place. By Lemma 3.11, we can obtain, in quadratic time, an execution sequence σ_3 such that $m_2 \stackrel{\sigma_3}{\to} m_3$ and m_3 is error marking. We have $\sigma = \sigma_1\sigma_2\sigma_3$. □

3.3.5. Error trace corresponding to an error pattern

In the three previous sections, we have proved that, for the each of the three error patterns described in Thm. 3.6, we are able to compute an error trace in quadratic time. The following theorem summarizes these results:

Theorem 3.17 (Identifying a pattern of Thm. 3.6 allows us to compute an error trace in quadratic time)**.** *Let $N = (P, T, F)$ be a free-choice workflow net.*

Given an error pattern described in Thm. 3.6, we can compute an error trace in quadratic time.

Proof. We consider the three error patterns separately:

1. If N contains a siphon that does not contain the initial place, by Thm. 3.14, we can compute an error trace in quadratic time.

2. If N contains a DQ siphon with a P/T handle, by Thm. 3.15, we can compute an error trace in quadratic time.

3. If N contains a simple path to the final place with a T/P handle, by Thm. 3.16, we can compute an error trace in quadratic time.

□

3.4. A structural characterization of soundness

Having established that we can derive an error trace from each error pattern of Thm. 3.6, we already have the proof for one direction of Thm. 3.6.

To prove the other direction, we will use the following theorem, initially presented by Kemper and Bause (41), which allows us to link soundness and structural soundness:

Theorem 3.18 (Soundness of a structurally sound workflow net (41))**.** *Let N be a free-choice workflow net such that $\overline{N}$ is structurally sound.*
N is sound if and only if all minimal siphons of $\overline{N}$ are marked by the initial marking.

In the following, we will prove two lemmas which link the notions of circuit with P/T handle without T/P bridge used by Esparza and Silva in their characterization of structurally live and bounded free-choice Petri nets with the notions of siphon with P/T handle and simple path to the final place with T/P handle that we use in our characterization.

In the proof of the first lemma, we will use the following theorem proved by Esparza and Silva (16) which shows that any transition in the subnet generated by a minimal siphon has a single pre-place.

Theorem 3.19 (Minimal siphon property (16))**.** *Let $N = (P, T, F)$ be a free-choice Petri net, $S \subset P$ be a siphon of N and $N' = (P', T', F')$ the subnet generated by S.*
S is minimal if and only if S is strongly connected and for every transition $t \in T'$, we have $|pre_{N'}(t)| \leq 1$.

The first lemma establishes that if there exists a circuit with a P/T handle without a T/P bridge in a connected workflow net, then the connected workflow net contains a siphon with a P/T handle as well:

Lemma 3.20. *Let N be a free-choice workflow net.*
If $\overline{N}$ contains a circuit with a P/T handle without a T/P bridge, then N contains a siphon with a P/T handle or a siphon that is not initially marked.

Proof. Let Ω be circuit of $\overline{N}$ and let H be a P/T handle of Ω without a T/P bridge. Let p be the place in $H \cap \Omega$ and t be the transition in $H \cap \Omega$.

Consider the minimal siphon S containing Ω. (We will present, in Sect. 3.5.3.1, Alg. 2 which allows us to compute a minimals siphon containing a given circuit. Therefore S exists.) There is a suffix π of H such that the first and last elements of π belong to S and all the other elements of π are disjoint from S: Note that t and p belong to the subnet generated by S. By Thm. 3.19, we have that $|\circ t \cap S| = 1$ and because $|\text{pre}_\Omega(t) \cap S| = 1$ and $\text{pre}_\Omega(t) \cap \text{pre}_H(t) = \emptyset$, we have $\text{pre}_H(t) \not\subset S$. Therefore, π exists and is non-trivial because it contains at least $\text{pre}_H(t)$ and t.

We now show that π is a P/T handle of S. By construction, π is disjoint from S aside from its first and last element and the last element of π is a transition. We are left to show that the first element x of π is a place. We proceed by contradiction: Suppose that x is a transition. By definition of siphon, S is strongly connected, therefore there is a simple path π' from x to an element p of Ω. Without loss of generality, we assume that p is the only element of π' that belongs to Ω. As $|\circ p \cup S| > 1$, p must be place by

definition of siphon. Thus π' is a T/P bridge from H to Ω which contradicts the initial assumption that H does not have a T/P bridge.

If S is not initially marked in $\overline{N}$ then S is also a non-initially marked siphon of N. If S is initially marked in $\overline{N}$, S contains the initial place of $\overline{N}$ and therefore, by definition of siphon (Def. 2.7), must contain the final place of $\overline{N}$. By definition of handle (Def. 3.1), H is disjoint from S aside from t and p and therefore H does not contain the return transition. Thus, S is also a siphon in N with a P/T handle.

□

The second lemma establishes that if a connected workflow net has a circuit with a T/P handle then the original workflow net has a simple path to the final place with a T/P handle:

Lemma 3.21. *Let N be a free-choice workflow net.*

If $\overline{N}$ has a circuit with a T/P handle then N has a simple path, from some element to the final place, with a T/P handle.

Proof. Let Ω be a circuit with a T/P handle H in $\overline{N}$ as illustrated by Fig. 3.9. Let t be the transition in $\Omega \cap H$ and p be the place in $\Omega \cup H$

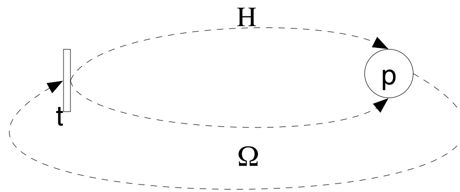


Figure 3.9.: A Circuit with a T/P handle.

Note the symmetry in the pattern: One can build a circuit Ω' by concatenating H and the portion of Ω from p to t. The portion of Ω from t to p is a T/P handle of Ω'.

We will show the existence of a simple path in N to the final place f of N with a T/P handle.

We distinguish two cases:

1. $f \in \Omega \cup H$: We can assume without loss of generality that $f \in \Omega$ because of the symmetry in the pattern. In $\overline{N}$, we have, by definition of connected workflow net, that $(f\circ)\circ$ contains only the initial place i of N. The portion of Ω from i to f is a simple path to f and H is a T/P handle of that path (See Fig. 3.10). Note that, while the figure illustrates a case where i is on the portion of Ω between t and p, this same reasoning applies in the case where i is on the portion of Ω between p and t.

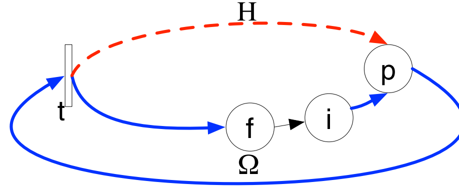


Figure 3.10.: Path with T/P handle case 1.

2. $f \notin \Omega \cup H$: Consider a simple path π' in N from p to f. Such a path exists by definition of a workflow net.

   Let x be the last element in $(\Omega \cup H) \cap \pi'$. We can assume without loss of generality that $x \in \Omega$ by the symmetry of the pattern.

   Let π^* be the path obtained by the concatenation of the portion of Ω from the element following x on Ω to x and the suffix of π' from x. π' is a simple path to f and H is a T/P handle of π^* (See Fig. 3.11). Again, the figure illustrates the most complex case where x is on the portion of Ω between t and p, the same reasoning applies in case x is on the portion of Ω between p and t.

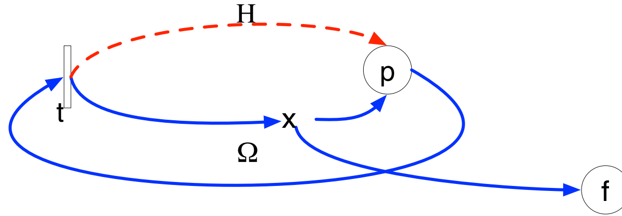


Figure 3.11.: Path with T/P handle case 2.

□

We can now prove our structural characterization of soundness:

Theorem 3.6 (Structural characterization of soundness)**.** *Let N be a free-choice workflow net.*

N is unsound iff one of the following statements holds:

1. *There exists a siphon that does not contain the initial place.*
2. *There exists a DQ siphon with a P/T handle.*[1]

[1]Strictly speaking this condition should be written: There exists a DQ siphon generating a subgraph which has a P/T handle.

3. *There exists a simple path from some element to the final place with a T/P handle.*

Proof. We show the two directions of the proof separately.

$\Leftarrow$ If N contains one of the error patterns, then N is not sound by Thm. 3.17.

$\Rightarrow$ By Thm. 3.18, N is sound if and only if $\overline{N}$ is structurally sound and satisfies there no siphon that does not contain the initial place. Therefore, we are left to show that if $\overline{N}$ is not structurally sound, then there exists a DQ siphon with a P/T handle or the exists a simple path from some element to the final place with a T/P handle:

Assume that $\overline{N}$ is not structurally sound. By Thm. 3.2, $\overline{N}$ contains a circuit with a T/P handle or a circuit with a P/T handle without T/P bridge. By Lemma 3.21, if $\overline{N}$ contains a circuit with a T/P handle, then N contains a simple path with a T/P handle. By Lemma 3.20, if there is circuit with a P/T handle without a T/P bridge in $\overline{N}$, then N either contains a siphon which is not initially marked, in which case we are done, or a siphon with a P/T bridge. If S is a siphon with a P/T bridge, S is DQ siphon, in which case we are done, or not. If S is not a DQ siphon, by DQ siphon definition Def. 3.4, there is transition $t \in \circ S$, such that $|t \circ \cap S| \geq 2$. We show later (in Thm. 3.26) that such transition allows us to derivate a circuit with a T/P handle in $\overline{N}$ which implies, by Lemma 3.20, that N contains a simple path from some element to the final place that has a T/P handle.

□

Relating this characterization to the examples presented informally in Sect. 3.2, the siphon highlighted in Fig. 3.3 does not contain the initial place p_1. The connected workflow net of Fig. 2.1 in Sect. 2.2 that we use as usual example does not violate the first property of the characterization but does violates the two others: the siphon highlighted in Fig. 3.4 has a P/T handle and the circuit highlighted in Fig. 3.5 has a T/P handle.

3.5. A polynomial technique to check the structural characterization

In this section, we describe a control-flow analysis which checks for the structural error patterns of Thm. 3.6.

We first present the general flow of the analysis and the structural diagnostic information provided by every step. Then we present how each step is achieved.

3.5.1. General flow of the analysis

The analysis we present in the following is performed on the connected version of the workflow net. If and only if the workflow net is unsound, the analysis identifies one of the three following patterns in the connected workflow net:

1. a siphon that does not contain the initial place,
2. a DQ siphon with a P/T handle, or
3. a circuit with a T/P handle.

The first two patterns are directly transferred to the error patterns in original workflow net. The circuit with T/P handle in the connected workflow net is transformed into a simple path to the final place with a T/P handle in the original workflow net using the construction described in Lemma 3.21.

This analysis uses the *rank theorem* from Desel, Esparza, and Silva (8, 12, 15) and extends the algorithm presented by Kemper and Bause (41) using the rank theorem to decide soundness.

The rank theorem allows us to check if a connected workflow net is structurally sound:

Theorem 3.22 (Rank theorem (12))**.** *Let $\overline{N}$ be a connected free-choice workflow net. Let M be its incidence matrix of $\overline{N}$ and k be its number of clusters.*

$\overline{N}$ is structurally sound if and only if

1. *$\overline{N}$ is SMD, and*
2. *$rank(M) = k - 1$.*

The first condition states $\overline{N}$ must be SMD, i.e., state machine decomposable (cf. Def. 2.7). We call the second condition of the Thm. 3.22 the *rank equation*. The rank equation requires the rank of the incidence matrix of $\overline{N}$ to be equal to the number of clusters of $\overline{N}$ minus one.

Fig. 3.12 illustrates the main flow of the structural control-flow analysis:

First we check for siphons that are not marked by the initial marking: if such a siphon is identified, we can directly return it as structural error.

Then we check that the connected workflow net is SMD. To do so we give an algorithm which attempts to compute a state machine decomposition. This algorithm can fail and, upon failure, returns either a siphon with a P/T handle or a circuit with a T/P handle. If the algorithm succeeds, a state machine decomposition is found and we know that the connected workflow net is bounded (9).

We can then check the rank equation and, if the equality holds, conclude that the connected workflow net and thus the workflow net is sound by Thm. 3.22 and Thm. 3.18. If the equality does not hold, we know that the connected workflow net is not live.

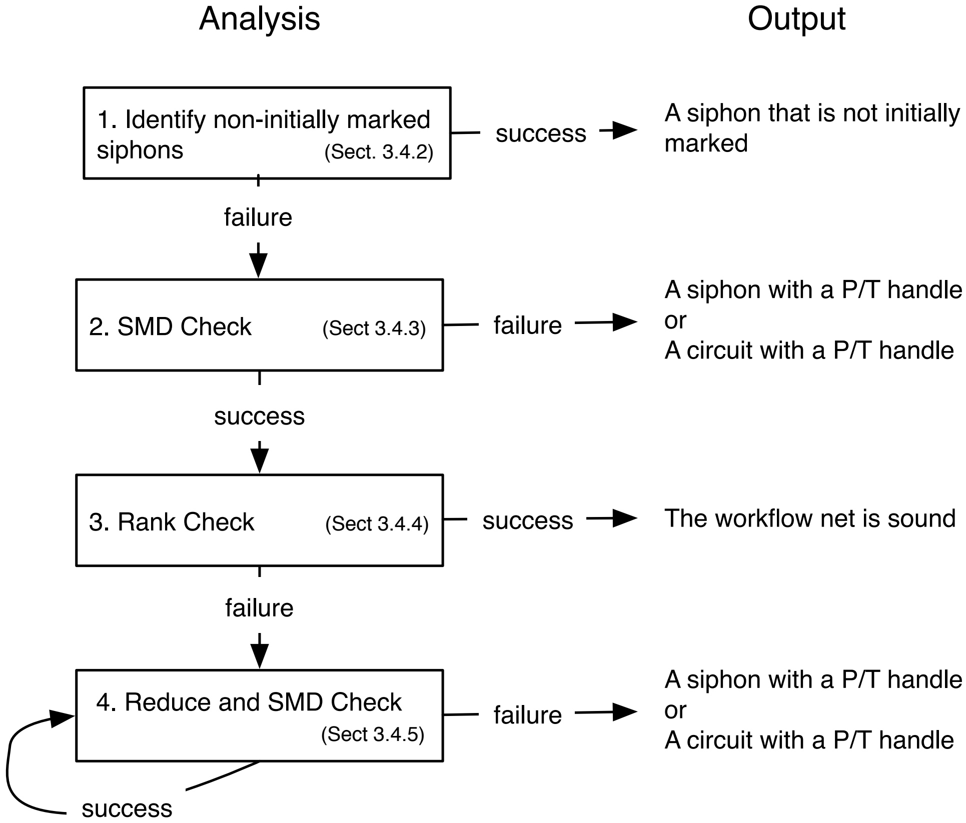


Figure 3.12.: The general flow of the analysis algorithm.

Unfortunately, the fact that the rank equation does not hold is not very useful as diagnostic information. Therefore we iteratively reduce the connected workflow and perform a state machine decomposition again at every iteration until we identify a structural violation. Such violation can be mapped to a violation in the original, i.e., non-reduced connected workflow net. The reduction is detailed in Sect. 3.5.5.2.

In the following, we will detail each step of the structural control-flow analysis, we will show that each step can be performed in polynomial time and that the reduction step always results in a smaller sub-net. This will prove that our approach runs in polynomial time.

3.5.2. Checking for siphons that are not initially marked

A siphon that is not initially marked such as, for example, the siphon highlighted on Fig. 3.3 is easily achieved using Alg. 1 This algorithm, proposed by Esparza (16) to compute a siphon that does not contain a set of places, allows us to check if a connected workflow net $\overline{N}$ contains a siphon which does not contain the initial place of $\overline{N}$.

Algorithm 1 findUnmarkedSiphon($\overline{N}$).
Input: a connected workflow net $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$
Output: a siphon of $\overline{N}$ which does not contain its initial place

Each node in $\overline{P} \cup \overline{T}$ is initially not visited.
Let p_0 be the initial place of $\overline{N}$.
We set p_0 to be visited.
A node $n \in \overline{P} \cup \overline{T}$ is *active* iff one node in $n\circ$ is not visited and all nodes in $\circ n$ are visited.
while There exists an active node n **do**
  Set each node in $n\circ$ to be visited.
end while
return The set of places which are not visited.

If the algorithm returns a siphon S such that $S \neq \emptyset$, then S is a violation of the first condition of our structural characterization of soundness Thm. 3.6 and an error trace can obtained in polynomial time based on S by Thm. 3.14.

3.5.3. Checking that the connected workflow net is structurally sound

We check that a connected workflow net is structurally sound by attempting to cover the connected workflow net by P-components (cf. Def. 2.7). As proposed by Kemper and Bause (41), we use the known result that minimal siphons generate P-components in structurally sound connected workflow nets:

Theorem 3.23 (Minimal siphons generate P-components in structurally sound connected workflow nets (2))**.** *Let $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ be a structurally sound connected workflow net.*

If $S \subseteq \overline{P}$ is a minimal siphon, then S generates a P-component.

We will see in Sect. 3.5.3.1 how one can compute a minimal siphon containing a given place in a connected workflow net. Therefore, we can easily obtain a set of minimal siphons covering a connected workflow net: We start by computing a siphon for a given place, let us say the initial place. Then, we compute another siphon containing a place that is not covered by any minimal siphon yet. We repeat the last step until all places are covered by a minimal siphon. By the contrapose of Thm. 3.23, we have: if a minimal siphon of N does not generate a P-component, then N is not SLB. Every time we compute a minimal siphon, we check that this siphon generates a P-component as we will describe in Sect. 3.5.3.2. The decomposition terminates either when we identified a siphon that is not a P-component or when all the places of the connected workflow net are covered by a P-component, i.e., when we have found a control-flow error or have established that the connected workflow net is SMD. In Sect. 3.5.3.3, we prove that, if a minimal siphon that does not generate a P-component, we can identify a violation of

the structural characterization of soundness in the form of a siphon with a P/T handle or circuit with a T/P handle.

3.5.3.1. Identifying minimal siphons

There exist a few algorithms in the literature to identify a minimal siphon containing a given place. We use Alg. 2 as presented by Esparza and Silva (16) and re-used by Kemper and Bause (41). This algorithm has a polynomial time complexity with respect to the size of the connected workflow net. Kemper (40) presented an improvement of this algorithm, which runs in linear time. For the sake of simplicity, we chose to base our presentation on the non-optimized version. The optimized algorithm could however be easily used instead.

Algorithm 2 findSiphonContaining(p, $\overline{N}$).
Input: a connected workflow net $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ and a place $p \in \overline{P}$.
Output: a siphon of $\overline{N}$ containing p.

Find a simple cycle c containing p
Let S be the set of places on c
while There exists a place p in the subnet generated by S such that $\circ p \setminus S \neq \emptyset$ **do**
  Let H be a handle of S containing a place $p' \in \circ p \setminus S$, and P_H be the set of places on H
  $S := S \cup E_H$
end while
return S

Alg. 2 returns a minimal siphon and ensures the following properties:

Lemma 3.24 (Properties of the siphon returned by Alg. 2)**.** *(16, 41)*
Invoking Alg. 2 on a connected workflow net $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ and a place $p \in \overline{P}$, results in a siphon S such that:

1. *The subnet $N' = (P', T', F')$ of $\overline{P}$ generated by S is a strongly connected subnet.*
2. *For each transition $t \in T'$, we have $|\circ t| = 1$.*
3. *For each place $p \in S$, we have $pre_{N'}(p) = pre_{\overline{N}}(p)$.*

3.5.3.2. Checking that a minimal siphon generates a P-component

A minimal siphon must satisfy the following properties in order to be a P-component:

Lemma 3.25 (Characterization for a minimal siphon to be a P-component (41))**.** *Let S be a minimal siphon of $\overline{N}$. The subnet N' generated by S is a P-component iff:*

- *For each transition $t \in \circ S$, $|t \circ \cap S| = 1$.*

- *For each place $p \in S$, $p\circ \subseteq \circ S$.*

It is easy to check these properties efficiently on a siphon. We say that a minimal siphon *generates a P-component* if it satisfies the conditions of Lemma 3.25.

3.5.3.3. Obtaining an error pattern from a failed SMD check

We now show that when a siphon does not satisfy one of the conditions to be a P-component, i.e., the siphon does not generate a P-component, we obtain an error pattern that is either a circuit with a T/P handle or a DQ siphon with P/T handle:

Theorem 3.26 (A failed SMD decomposition provides structural diagnostic information)**.** *Let $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ be a connected free-choice workflow net.*

When a minimal siphon S of $\overline{N}$ does not generate a P-component, we can compute a circuit with a T/P handle or a siphon with a P/T handle.

Proof. Let $N' = (P', T', F')$ be the subnet of $\overline{N}$ generated by S. As N' is not a P-component, by Lemma 3.25, we have one these two violations:

1. There is a transition $t \in \circ S$, such that $|t \circ \cap S| \geq 2$:

   We show that, in this case, there exists (in N') a circuit with a T/P-handle:

   Consider two places $p_1, p_2 \in t \circ \cap P'$. Because N' is strongly connected, there exists two circuits $\Omega =< t, p_1, \ldots, t >$ and $\Omega' =< t, p_2, \ldots, t >$ in N'. Consider the prefix H of Ω' such that H only intersects with Ω with its first node t and its last node n. By Lemma 3.24.3, no transition in N' has more than one incoming arc in N'. Thus n is a place because it has at least two incoming arcs in N'. We have shown that H is a T/P-handle of Ω.

2. There exists a place p such that a transition in $p\circ$ does not belong to $\circ S$:

   We show that N' has a P/T-handle:

   Because $\overline{N}$ is strongly connected there exists a circuit $\Omega' =< p, t, \ldots, p >$ in $\overline{N}$. Let H be the prefix of Ω' from p to the first node n on Ω' after p such that $n \in P' \cup T'$. (i.e., H is a path in N disjoint from N' aside from p and n). Thus H is an handle of N'. The element n is a transition because, if n was a place, all the transitions in $\circ n$ would belong to N' by Lemma 3.24 and therefore n would not be the first node of h after p in N'.

   Thus H is P/T-handle of N'.

□

Note that, as discussed in Sect. 3.5.1, a circuit with T/P handle and a DQ-siphon with a P/T handle in a connected workflow net correspond to a simple path to the final place with a T/P handle and a DQ-siphon with a P/T handle in the original workflow net, respectively.

3.5.4. Checking the rank equation

Once we established that the connected workflow net is SMD, we check that the rank equation holds. If that is the case, the connected workflow net and thus the original workflow net is sound. If it is not the case, the connected workflow net is not structurally live, i.e., there exists no live initial marking.

Checking that the rank equation holds involves computing the rank of the incidence matrix of the connected workflow net and to compute the number of clusters of the connected workflow net. Computing the number of clusters requires linear time with respect to the size of the connected workflow net. Well-established algorithms (45) can be used to compute the rank of the incidence matrix in cubic time with respect to the size of the matrix. Therefore checking the rank equation can be performed in cubic time with respect to the size of connected the workflow net.

3.5.5. When the rank equation check fails

In Sect. 3.5.5.1, we present an example of an unsound SMD connected workflow net, i.e., an unsound connected workflow net for which a state machine decomposition succeeds. Our approach in this case is the following: We reduce the connected workflow net as described below and then perform a new state machine decomposition on the reduced connected workflow net. If the decomposition succeeds, we repeat the reduction and state machine decomposition steps. These steps are performed until the decomposition fails.

In Sect. 3.5.5.2, we will present two algorithms that allow us to perform a reduction of an unsound connected workflow net which always results in a smaller but still unsound subnet. In Sect. 3.5.5.3, we prove that when the SMD check identifies a siphon S that does not generate a P-component on a reduced connected workflow net obtained using the algorithms of Sect. 3.5.5.2, the siphon S also exists in the original,i.e., non-reduced, connected workflow net and S also does not generate a P-component in the original connected workflow net. In Sect. 3.5.5.4, we then show that the algorithms of Sect. 3.5.5.2 reduce, i.e., remove at least one place, of any unsound SMD connected workflow net. As the connected workflow net is finite, there is a finite number of reduction steps. Moreover, as the reduction preserves the unsoundness, the state machine decomposition will eventually identify a siphon with a P/T handle or a circuit with a T/P handle.

3.5.5.1. An unsound connected workflow net for which a state machine decomposition exists

Fig. 3.13 illustrates an unsound connected workflow net which is covered by the P-component generated by the siphons $\{i, p_1, p_3, p_4, f\}$ and $\{i, p_2, p_5, p_6, f\}$. As it is unsound, the rank equation fails and we can conclude that the connected workflow net is not structurally live, but we do not have any good diagnostic information. In the

following, we describe how we iteratively reduce and perform another state machine decomposition in order to obtain a structural violation.

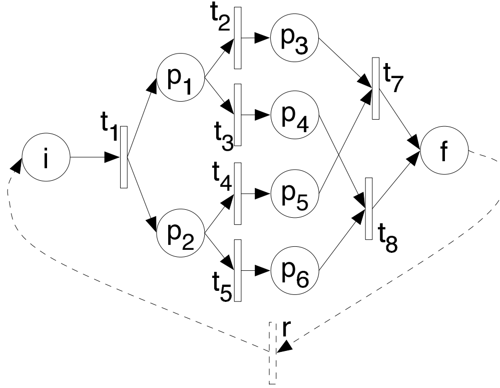


Figure 3.13.: A SMD connected workflow net containing a deadlock.

Side note: The connected workflow net illustrated by Fig. 2.1, which we have used as main example so far, is not SMD. Also note that Alg. 2 is non-deterministic, and the state machine decomposition of the connected workflow net illustrated by Fig. 3.13 has only 1 chance out of 6 to be decomposed into P-components without finding a violation. For example, the minimal siphon $\{i, p_1, p_2, p_3, p_6, f\}$ can be returned by Alg. 2. This would allow us to identify the circuit $< i, t_1, p_2, t_5, p_6, t_8, f, r, i >$ and its T/P handle $< t_1, p_1, t_2, p_3, t_7, f >$. We suspect that, in practice, there will be very few unsound connected workflow nets for which we find a state machine decomposition. This intuition will be confirmed by our experiments in Chap. 7.

3.5.5.2. Reducing the connected workflow net

We first present two simple algorithms which will allow us to obtain a reduced connected workflow net from an unsound SMD connected workflow net. A reduction step involves the successive application of Alg. 3 and Alg. 4 which are detailed below.

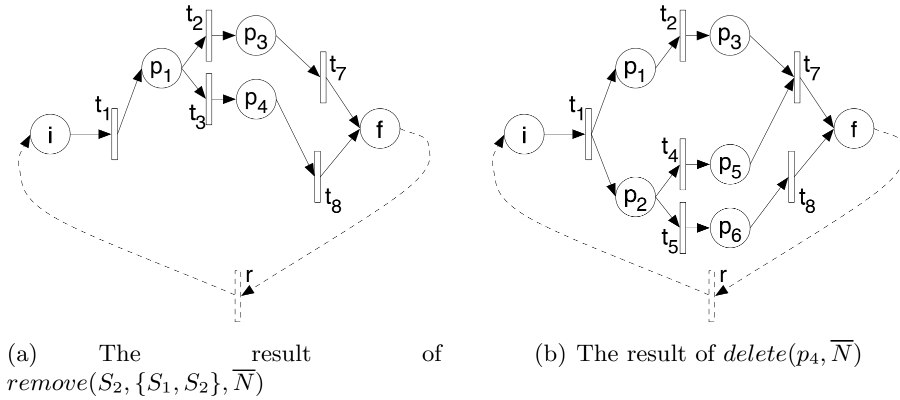
We start by defining two reduction functions that are used by the reduction algorithms. Intuitively, the first function $remove(S, \mathcal{S}, \overline{\mathrm{N}})$ removes the entire subnet that is covered by the P-component S and not by any other P-component in $\mathcal{S}$ from the net $\overline{N}$. The second function $delete(p, \overline{N})$ deletes the place p from the net $\overline{N}$ and a few more elements to ensure that the resulting net remains strongly connected. More formally:

Definition 3.27. *Let $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ be a connected free-choice workflow net. Let $\mathcal{S}$ be a set of P-components that cover $\overline{N}$. Let $S \in \mathcal{S}$ and $p \in \overline{P}$.*

- *The function remove*$(S, \mathcal{S}, \overline{N})$ *returns maximal the subnet of* $\overline{N}$ *that is covered by the P-components in* $\mathcal{S} \setminus \{S\}$.

- *The function delete*$(p, \overline{N})$ *returns the maximal subnet* $N' = (P', T', F') \subset \overline{N}$ *such that for each element* $e \in T' \cup P'$ *there exists a circuit in* $\overline{N}$ *that contains* e *but not* p.

Let $\overline{N}$ be the connected workflow net illustrated by Fig. 3.13. The set $\mathcal{S} = \{S_1, S_2\}$ where $S_1 = \{i, t_1, p_1, t_2, t_3, p_3, p_4, t_7, t_8, f, r\}$ and $S_2 = \{i, t_1, p_2, t_4, t_5, p_5, p_6, t_7, t_8, f, r\}$ is a state machine decomposition of $\overline{N}$. Fig. 3.14(a) illustrates the result of applying $remove(S_2, \mathcal{S})$ and Fig. 3.14(b) illustrates the results of applying $delete(p_4, \overline{N})$.



(a) The result of $remove(S_2, \{S_1, S_2\}, \overline{N})$

(b) The result of $delete(p_4, \overline{N})$

Figure 3.14.: The result of the two reduction functions on the connected workflow net $\overline{N}$ illustrated by Fig. 3.13.

Alg. 3 is not strictly necessary in order to reduce the connected workflow net but it allows us to obtain a smaller set of P-components containing a control-flow error efficiently. We expect it to speed up the reduction in practice. The idea behind Alg. 3 is to iterate through the P-components obtained by a state machine decomposition and for each P-component, to check whether removing the P-component results in a sound connected workflow net. If the connected workflow net remains unsound, we remove the P-component. The result is a set of P-components such that none of the element of the set can be removed without 'removing' the control-flow error. We say that the set of P-component returned by Alg. 3 is a *minimal* set of P-components generating the error.

Similarly to Alg. 3, Alg. 4 reduces the connected workflow net by removing a portion of a workflow. Alg. 4, also uses the rank equation to check that it is removing a sensible portion of the connected workflow net. The key difference between Alg. 3 and Alg. 4 is that Alg. 4 does not remove an entire P-component of the net but a carefully computed subset of it. In the following section, we will prove that we can remove such a portion as

Algorithm 3 findMinimalSetPComp($\overline{N}$, $\mathcal{S}$).
Input: An unsound connected workflow net $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ and a set $\mathcal{S}$ of P-components covering $\overline{N}$.
Output: A minimal set $\mathcal{S}' \subseteq \mathcal{S}$ of P-components generating the error.

```
Let S' = S and removed be a boolean variable
repeat
    removed = false
    for each P-component S ∈ S' do
        Let N' be the subnet returned by remove(S, S', N̄)
        if rank(N') ≠ (number of clusters of N') −1 then
            S' = S' \{S}
            removed = true
        end if
    end for
until ¬ removed
return S'
```

long as no error is identified and therefore that we can reduce the connected workflow net until we obtain an error pattern.

Side note: We suspect that Alg. 3 always manages to reduce the number of P-components of an unsound SMD connected workflow net to two. This significantly reduces the number of iterations of the outer loop of Alg. 4.

Algorithm 4 reduce($\mathcal{S}$).
Input: A set $\mathcal{S}$ of P-components covering an unsound connected workflow net $\overline{N}$.
Output: A strongly connected subnet of $\overline{N}$.

```
for each P-component S ∈ S do
    Let P_s be the set of places in ∘S such that for each p ∈ P_s, p ∉ S
    for each place p ∈ P_s do
        Let N̄' be the subnet returned by delete(p, N̄)
        if rank(N̄') ≠ (number of clusters of N̄') −1 then
            N̄ = N̄'
        end if
    end for
end for
return N̄
```

Consider the connected workflow net illustrated by Fig. 3.13 again and its decomposition $\mathcal{S}$ into the previously defined P-components S_1 and S_2. We will discuss how the

two algorithms described above would allow us to reduce this connected workflow net in order to obtain a subnet that is not SMD anymore: Removing any P-component with the remove function would not preserve the unsoundness (see, for example, the result of illustrated by Fig. 3.14(a)). Therefore Alg. 3 would not remove any P-component. On the other hand Alg. 4 would delete some elements of the net and return a smaller net: Let $S = S_2$ in the first iteration of the outer loop of *Alg.* 4, then $P_s = \{p_3, p_4\}$. Let $p = p_4$ be in the first iteration of the outer loop of *Alg.* 4, then we would obtain the subnet $\overline{N}'$ illustrated by Fig. 3.14(b), by the application of $delete(p, \overline{N})$. The subnet $\overline{N}'$ preserves the unsoundness, i.e., we have that rank($\overline{N}'$) $\neq$ (number of clusters of $\overline{N}'$) -1. The subnet $\overline{N}'$ will not be reduced further by Alg. 4 as deleting p_3 using the delete function would result in the subnet S_2 itself which would remove the unsoundness. Algorithm 4 would therefore return the subnet illustrated by Fig. 3.14(b). This subnet is not SMD, i.e., it cannot be decomposed into P-component. Therefore the next step of the analysis will identify a structural error.

3.5.5.3. Unsoundness monotonicity

The first step in proving the correctness of our reduction approach is to show that an error in the reduced connected workflow net is also an error of the original connected workflow net. We start by showing that a siphon of the reduced net is also a siphon of the original net.

Lemma 3.28. *Let $\overline{N}$ be an unsound connected free-choice workflow net let $\overline{N}'$ be the strongly connected free-choice Petri net obtained by using Alg. 3 or Alg. 4.*
If S is a siphon of $\overline{N}'$, we have:

1. $pre_{\overline{N}}(S) = pre_{\overline{N}'}(S)$.
2. *and* $post_{\overline{N}'}(S) \subseteq post_{\overline{N}}(S)$.
3. *S is also a siphon of $\overline{N}$.*

Proof. By construction, S is also a subset of the places of $\overline{N}$.

1. $\text{pre}_{\overline{N}}(S) = \text{pre}_{\overline{N}'}(S)$ because Alg. 3 and Alg. 4 never remove a transition without removing its target place(s) first. Therefore any place remaining in $\overline{N}'$ has the same set of incoming transitions as in $\overline{N}$.
2. $\text{post}_{\overline{N}'}(S) \subseteq \text{post}_{\overline{N}}(S)$ because Alg. 3 and Alg. 4 only remove elements of the connected workflow net.
3. By definition of siphon, $\text{pre}_{\overline{N}'}(S) \subseteq \text{post}_{\overline{N}'}(S)$. By 1 and 2, $\text{pre}_{\overline{N}}(S) \subseteq \text{post}_{\overline{N}'}(S) \subseteq \text{post}_{\overline{N}}(S)$ and therefore S is a siphon of $\overline{N}$.

□

We can now prove that a siphon that does not generate a P-component in the reduced net, i.e., a siphon that points to a violation of the structural characterization, does not generate a P-component in the original net:

Theorem 3.29. *Let $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ be an unsound connected free-choice workflow net, let $\mathcal{S}$ be a set of P-components covering $\overline{N}$, and let $\overline{N}' = (\overline{P'}, \overline{T'}, \overline{F'})$ be the strongly connected free-choice Petri net obtained by using Alg. 3 or Alg. 4.*

If a minimal siphon $S \in \mathcal{S}$ does not generate a P-component in $\overline{N}'$, S is a minimal siphon of $\overline{N}$ which does not generate a P-component.

Proof. Let $S \in \mathcal{S}$ be a siphon of $\overline{N}'$ such that S does not generate a P-component. By Lemma 3.28.3, S is a siphon of $\overline{N}$ as well. We distinguish two cases:

1. There exists a transition $t \in \mathrm{pre}_{\overline{N}'}(S)$, such that $|\mathrm{post}_{\overline{N}'}(t) \cap S| \geq 2$. By Lemma 3.28.1, $\mathrm{pre}_{\overline{N}'}(S) = \mathrm{pre}_{\overline{N}}(S)$ and therefore $t \in \mathrm{pre}_{\overline{N}}(S)$. The transition $t \in \mathrm{pre}_{\overline{N}}(S)$ is such that $|\mathrm{post}_{\overline{N}}(t) \cap S| \geq 2$.

2. There exists a transition $t \in \mathrm{post}_{\overline{N}'}(S)$, such that $t \notin \mathrm{pre}_{\overline{N}'}(S)$. By Lemma 3.28.2, $\mathrm{post}_{\overline{N}'}(S) \subseteq \mathrm{post}_{\overline{N}}(S)$ and therefore $t \in \mathrm{post}_{\overline{N}}(S)$. By Lemma 3.28.1, $\mathrm{pre}_{\overline{N}'}(S) = \mathrm{pre}_{\overline{N}}(S)$ and therefore $t \notin \mathrm{pre}_{\overline{N}}(S)$.

□

3.5.5.4. An unsound SMD connected workflow net is always reduced by Alg. 4

We now show that whenever there is a SMD connected workflow net that is unsound, it can always be reduced by Alg. 4. We proceed using two lemmas showing that *Alg.* 4 removes at least one place of a strongly connected free-choice Petri net N that is not structurally sound, i.e., if there is a circuit with a T/P handle in N or there is a circuit with a P/T handle without a T/P bridge in N, respectively:

Lemma 3.30. *Let N be strongly connected free-choice Petri net. Let $\mathcal{S}$ be a set of P-components covering N.*

If there is a circuit with a T/P handle in N, then Alg. 4 removes at least one place of N.

Proof. Let Ω be a circuit with a T/P handle H in N (cf. Fig. 3.15). Let t be the first transition of H and p the last place of H. Let $p_1 \in t \circ \cap H$ and $p_2 \in t \circ \cap \Omega$. As t is a transition and $p_1, p_2 \in t\circ$, by definition of P-component p_1 and p_2 must belong to a different P-component of $\mathcal{S}$. Let $S_1 \in \mathcal{S}$ be a P-component which covers p_1. Let t' be the first element following Ω backward from t such that $\mathrm{pre}_\Omega(t') \cap \mathrm{pre}_{S_1}(t') = \emptyset$. The element t' exists because $p_2 \notin S_1$.

Let $S_2 \in \mathcal{S}$ be a P-component covering the element in $\mathrm{pre}_\Omega(t')$. Consider the iteration of the outer loop of Alg. 4 of which S_2 is the iteration variable. As $\mathrm{pre}_{S_1}(t') \neq \mathrm{pre}_{S_2}(t')$ and $t' \in S_1 \cap S_2$, by definition of P-component, the element t' must be a transition. As

P-components are strongly connected, the set $\circ t'$ contains a place $p' \in S_1$ and a place $p'' \in S_2$ such that $p' \neq p''$.

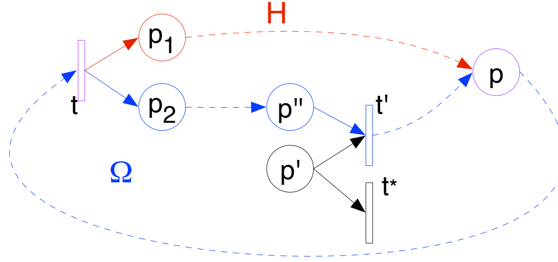


Figure 3.15.: Illustration for the proof of Lemma 3.30.

We now show that Alg. 4 removes the place p' by showing that $p' \notin H \cup \Omega$ which implies that removing p' would preserve Ω and H. We proceed by contradiction: Suppose that $p' \in H \cup \Omega$. By construction, the arc from p' to t' cannot be in Ω. Moreover, the arc from p' to t' cannot be in H because the only transition of H which is on Ω is t and it is clear that $t \neq t'$. Therefore, there must be another arc succeeding p' on H resp. Ω. Hence, there should be a transition $t^* \in \text{post}_{H\cup\Omega}(p')$ such that $t^* \neq t'$ and as $|\circ t'| > 1$ this would contradict the assumption that N is free-choice. ☐

Lemma 3.31. *Let N be strongly connected free-choice Petri net. Let $\mathcal{S}$ be a set of P-components covering N.*

If there is a circuit with a P/T handle without a T/P bridge in N, then Alg. 4 removes at least one place of N.

Proof. Assume that there exists a circuit Ω with a P/T handle H without a T/P bridge in N. Let p be the place in $\Omega \cap H$. Let t be the transition in $\Omega \cap H$.

We consider the P-component $S \in \mathcal{S}$ that covers the place $p_1 \in \text{pre}_\Omega(t)$. Note that the definition of P-component implies that S must cover t and the place $p_2 \in \text{pre}_\Omega(t)$ is not covered by S. We first show that either S covers Ω or a place is removed by Alg. 4. We proceed by contradiction:

- Suppose that there is an element $x \in \Omega$ that is not covered by S let t' be the first element after x on Ω which is covered by S. We know that such element exists because p is covered by S. Because $\text{pre}_\Omega(t) \not\subset S$ and $t \in S$, by definition of P-component, t is a transition. Let p' be the place such that $p' \in \text{pre}_\Omega(t)$. Because S is strongly connected, there exists a place $p^* \in \text{pre}_S(t)$. By construction $p^* \neq p'$. Because N is covered by $\mathcal{S}$, there exists a siphon $S' \in \mathcal{S}$ covering p'.

Alg. 4 removes p^* when S' is picked as master and if and only if the resulting net remains unsound. We show that $p^* \notin H \cup \Omega$ and therefore removing p^* would result in a net which contains $\Omega \cup H$ and is therefore unsound. We proceed by contradiction:

- Suppose that $p^* \in \Omega \cup H$. We distinguish two cases:
  1. If $p^* \in \Omega$, the arc between p^* and t' cannot belong to Ω by definition of circuit. Because Ω is a circuit there is a transition t^* in $\text{post}_{\Omega}(p^*)$. Because the arc between p^* and t' does not belong to Ω, $t^* \neq t'$.
  2. If $p^* \in H$, the arc between p^* and t' cannot belong to H because handle does not intersect with Ω aside from p and t and $t' \neq t$ by construction. Because H ends with a place, there must be a transition t^* in $\text{post}_H(p^*)$. Because the arc between p^* and t' does not belong to Ω, $t^* \neq t'$.

  In both cases the subnet containing p', t', p^*, t^*, and their connecting arcs contradict the assumption that N is free-choice.

Therefore p^* does not belong to $H \cup \Omega$ and is removed by Alg. 4.

We are left with the case where Ω is covered by S. Let t' be the last element of $H \setminus \{t\}$ covered by S. The element t' exists because $p \in S$ and $p_2 \notin S$. The same argumentation as above can be used to show that either all element of H preceding t' belong to S or there exists a place that is removed by Alg. 4. Therefore, we are left with the case where Ω and all the element of H until t' are covered by S. Let π be a simple path in S from t' to p. Such path exists because S is strongly connected. Let p^* be the first element of π such that $p^* \in H \cup \Omega$. We distinguish two cases.

1. $p^* \in H$. By construction, p^* must precede t' on H because there is no element in $H \setminus \{p\}$ that is covered by S by definition of t'. By definition of P-component, the element p^* is a place because $\text{pre}_S(p^*)| > 1$. Figure 3.16 illustrates the main elements of this portion of the proof. Let Ω^* be the circuit obtained by the concatenation of H and the portion of Ω from p to t. Let H^* be the prefix of π until p^*. H^* is a T/P handle of Ω^*. By Lemma 3.30, Alg. 4 removes a place of N.

2. $p^* \in \Omega$. We rule out this case by contradiction: By definition of P-component, the element p^* is a place because $\text{pre}_S(p^*)| > 1$. The prefix of π until p^* is a T/P bridge from H to Ω which is ruled out by assumption.

□

We can now prove that Alg. 4 reduces any unsound SMD strongly connected free-choice Petri net:

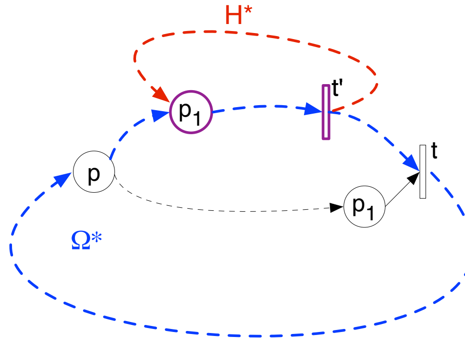


Figure 3.16.: The path H^* is a T/P handle of Ω^*.

Theorem 3.32 (Alg. 4 reduces any unsound SMD strongly connected free-choice Petri net)**.** *Let N be a strongly connected free-choice Petri net such that N is SMD and is not structurally sound.*

Applying Alg. 4 to N removes at least one place of N.

Proof. By Thm. 3.2, N contains a circuit with a T/P handle or a circuit with a P/T handle without a T/P bridge.

If N contains a circuit with a T/P handle, then *Alg.* 4 removes at least one place of N by Lemma 3.30.

If N contains a circuit with a circuit with a P/T handle without a T/P bridge, then *Alg.* 4 removes at least one place of N by Lemma 3.31. □

By Alg. 4 we have that Alg. 4 always reduces any unsound SMD strongly connected free-choice Petri net into a smaller unsound strongly connected free-choice Petri net. Moreover, by Thm. 3.29, we have that any net reduced by Alg. 4 and Alg. 3 retains the unsoundness and that the analysis will eventually identify a violation of Thm. 3.6. As the nets that we analyze are finite, this implies that the control-flow analysis algorithm presented in this chapter will always terminate and, as every step is polynomial, it will terminate in polynomial time. In conclusion, we can formulate the summarizing theorem:

Theorem 3.33 (The control-flow analysis is complete, correct, and polynomial)**.** *Let N be a free-choice workflow net.*

The analysis described by Fig. 3.12 returns a structural pattern in N violating Thm. 3.6 if N is unsound, it returns a message indicating that N is sound otherwise.

The analysis described by Fig. 3.12 terminates in polynomial time with respect to the size of N.

3.6. Summary

In this chapter, we presented a structural characterization of soundness and showed how to check it in polynomial time. This technique provides two types of diagnostic information: a structural pattern in the workflow net and a reduced error trace. Both diagnostic pieces of information are obtained in polynomial time. We believe that an error trace, which exhibits an erroneous execution, will easily convince an non-expert user that a business process model contains a control-flow error. On the other hand the structural error pattern will need some training, possibly by the visualization of error traces, of the user to be recognized directly as an error but such a pattern will also lead to a deeper understanding of the cause of the error and will also allow us to provide better fix suggestions. As discussed in the introduction, an error trace is a symptom of an error but not its cause. We therefore envision the two displays to be complementary for the user to locate, understand, and fix an control-flow error. We will discuss the user interaction and the fix suggestions that we can provide based on the diagnostic information in Chap. 6.

3.7. Related work on structural analysis

As already mentioned, this chapter builds on two main pieces of work: The first is the characterization of soundness by Esparza and Silva (14) (Thm. 3.2). This characterization is structural and would be very likely to lead to a valuable diagnostic information. Unfortunately, there exists no known approach to check it in polynomial time. The second is the work from Kemper and Bause (41) who provide an algorithm to check soundness in polynomial time. Kemper (40) even refines one part of the approach to make the overall check have a cubic time complexity with respect to the workflow net. Unfortunately, there is no diagnostic information directly available from this technique.

Another approach to check structural soundness has been presented by Esparza (13). Esparza provides a set of reductions rules for free-choice Petri nets. The Petri net is sound if and only if it can be reduced to the empty net. These reductions allow one to verify soundness in polynomial, but more than cubic, time. Although this approach could provide some error information, it has not been worked out yet. Similarly, Sadiq and Orlowska (57) propose reduction rules for workflow graphs (without IOR-joins). These rules are unfortunately not complete, i.e., some irreducible workflow graphs are sound.

Van der Aalst (66) has shown that, the absence of some type of handle in a free-choice net is a sufficient condition to ensure that it is structurally sound. He points out that path with handles can be computed using a maximum flow approach. Various algorithms exist to compute the maximum flow (see (31) for a list). The complexity of these algorithms ranges between $O(|N| \cdot |E|^2)$ and $O(|N| \cdot |E| \cdot log(|N|))$. The existence of a handle can be checked by applying a maximum flow algorithm to each pair of

transition and place of the net. Therefore, the complexity of detecting handles with such an approach is at best $O(|N|^3 \cdot |E| \cdot log(|N|)$. While one can check for these handles and they offer structural diagnostic information, the main drawback of this technique is that the absence of such handle is not a necessary condition to ensure that the free-choice net is structurally sound, i.e., the technique cannot be used to check structural soundness.

As discussed in Sect. 1.3, numerous alternative approaches to check for control-flow errors are based on state space exploration. We will discuss them in Sect. 6.7.

4

Replacing Inclusive OR logic

The business processes are modeled using industrial languages which are more complex than the control-flow models that are typically used for their analysis. The coverage of an analysis technique is directly linked to how well a business process model control-flow can be translated into the control-flow model on which the analysis is based.

In this chapter, we study the question to what extent a process model with IOR logic, in particular IOR-join, can be translated into a control-flow model without IOR logic.

More precisely, we focus on the control flow of a business process, which is modeled by a workflow graph. We ask whether a workflow graph with IOR logic can be translated into a workflow graph with only XOR and AND gateways.

We introduce the problem in more details in Sect. 4.1. In Sect. 4.2, we describe the requirements of the desired translation. In Sect. 4.3, we present three approaches to translate IOR-splits. In Sect. 4.4, we consider *local replacements* of IOR-joins. This translation strategy consists of replacing an IOR-join by a *partial workflow graph* that connects to the edges left dangling by the removal of the IOR-join. Such a local replacement fully maintains, apart from the IOR-join, the original workflow graph and thus makes the mapping to the original workflow graph trivial. Moreover, it leads to a very intuitive notion of equivalence: the partial workflow graph must have the same behavior as the IOR-join. We characterize under which conditions a local replacement is possible in an acyclic workflow graph and define a replacement for these cases. In Sect. 4.5, we consider a non-local translation strategy and characterize its condition of application. A non-local replacement essentially still retains the structure of the original workflow graph and allows us to replace some IOR-joins that have no local replacement. In Sect. 4.6, we relax our notion of replacement even more, and we show

that even then, there exist simple acyclic workflow graphs that have IOR-joins that cannot be replaced. In Sect. 4.7, we discuss our result and its implications to the translations of workflow graphs containing IOR-joins into Petri nets. We summarize our results in Sect. 4.8 and relate them to the existing literature in Sect. 4.9.

A first version of these results has been presented at the 10th International Conference on Business Process management (BPM 2012) (27). An extended version of the results was published in the Information System journal (22).

4.1. Problem

Business processes are in practice often modeled in industrial languages such as BPMN 2.0 (54) whereas many analysis techniques, such as the control-flow analysis described earlier, are based on workflow nets. One major obstacle to translate process models between industrial languages and workflow nets is the presence of gateways with inclusive OR (IOR) logic in the industrial languages.

An IOR gateway forks or synchronizes a variable set of flows, thereby supporting various workflow patterns (68). Analyses which do not support IOR gateways are therefore limited in coverage.

There are two aspects of the IOR gateway: the split and the join. On the one hand, the IOR-split has a *local* semantics, i.e., the enablement and effect of a transition executing an IOR-split relates only to its adjacent places. We will present a few easy translations of the IOR-split. On the other hand, the IOR-join, which has a *non-local* semantics, i.e., the enablement of the IOR-join depends on the marking of the whole workflow graph, is more difficult to translate. [1]

We will only deal with acyclic workflow graphs, for which the IOR-join semantics is simpler than in a cyclic context. This will allow us to provide replacement strategies for IOR-joins. Conversely, it will already suffice to display simple workflow graphs in which, in some formal sense, an IOR-join cannot be replaced. This will reveal an intrinsic limitation on the replaceability of IOR-joins and hence the translatability of the workflow graph of general process models into Petri nets.

Note that best practices suggest using IOR-joins in acyclic contexts only. The language BPEL (39) even restricts the use of IOR-joins to acyclic processes. Furthermore the replacement techniques for acyclic graphs can also be applied to cyclic graphs in case the IOR-joins can be separated in certain acyclic *single-entry-single-exit fragments* using graph parsing techniques (56, 74), cf. also (80).

[1] A special case of this problem arises when we want to translate a free-choice workflow net with multiple sinks (or equivalently, a workflow graph without inclusive logic with multiple sinks) into a free-choice workflow net with a unique sink. This is because the termination semantics for multiple sinks is usually equivalent with the IOR-join semantics. Note that a workflow net or -graph has a unique sink in the classical definition.

4.2. Translation requirements

The requirements of a translation from a workflow graph with IOR-joins into a workflow graph without IOR-joins can vary for different use cases. To obtain general, yet useful results, we take the following requirements into account:

- The translated workflow graph must have *equivalent* behavior. Many notions of equivalence exist (71). The adequacy of an equivalence for the translation depends on the use case. We will present the equivalences we use later in the chapter. Note that this requirement is by itself not challenging as one can easily 'unfold' an acyclic workflow graph into its finite full behavior (i.e., computation tree) and then encode this 'unfolding' as a sequential workflow graph. Such a construction would preserve, depending on its precise execution, many popular behavioral equivalences, such as trace equivalence and bisimulation. However, the obtained translation is in general exponentially larger than the original workflow graph. Therefore, we require that

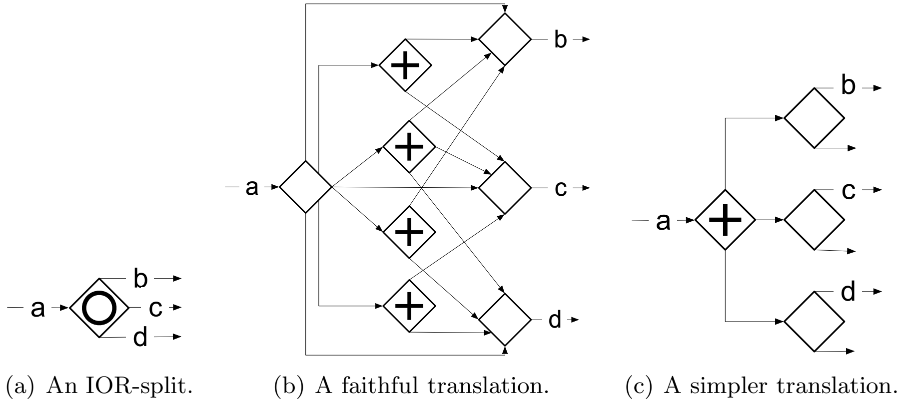
- the increase in size of the obtained workflow graph must be manageable: While the definition of a strict bound for a manageable increase depends on the use case, an exponential blowup is usually not acceptable. As of today, we are not aware of any general translation from workflow graphs with IOR-joins into Petri nets that preserves the behavior and does not incur an exponential blowup.

- Furthermore, we are interested to preserve the structure of the workflow graph: There should be a way to map the elements of the translated workflow graph back to the elements of the original workflow graph. This requirement is important if we want to map analysis results between the two workflow graphs. For example, in order to return to the user of an analysis technique the results in terms of the original process model or, when monitoring or administrating a process, to understand a trace or a state of the running process in terms of the original process model.

4.3. Translating IOR-splits

By definition (Def. 2.18), an IOR-split consumes a token from its single incoming edge, and it produces a token on each of a non-empty subset of its outgoing edges. This semantics is local and it can be translated to XOR- and AND-logic as shown in Fig. 4.1(b).

The replacement illustrated by Fig. 4.1(b), replacing the IOR-split from Fig. 4.1(a), creates an XOR-split and a branch for each possible assignment of the outgoing edges of the IOR-split. This replacement faithfully mimics the semantics of the IOR-split and fully preserves the structure of the original workflow graph. Unfortunately, the resulting workflow graph grows exponentially with respect to the number of outgoing edges of the IOR-join.



(a) An IOR-split. (b) A faithful translation. (c) A simpler translation.

Figure 4.1.: Two possible ways to replace an IOR-split.

Another translation for an IOR-split is shown in Fig. 4.1(c). This replacement only adds two edges and one gateway per outgoing edge. However, this replacement has two drawbacks: 1. It creates an additional sink per outgoing edge, which can be a problem for use cases requiring a single sink. 2. It does not enforce that at least one outgoing edge carries a token. In some use cases, this can be enforced by the user through the specification of data-related conditions specifying when each outgoing edge carries a token. Therefore, for a use case like translating a BPMN 2.0 process to be executed on an execution engine that does not support IOR-logic directly, this replacement can be adequate. For use cases where the data-related conditions are abstracted, like most control-flow analysis, this replacement is not adequate.

However, both ideas, from Fig. 4.1(b) and from Fig. 4.1(c), can be combined into a third translation, which is shown in Fig. 4.2. There, a first decision makes sure that at least one of the outcomes b, c, or d takes place, and afterwards, a set of pairwise concurrent decisions chooses additional optional outcomes. This is a faithful translation where the number of gateways is linear and the number of edges is quadratic in the number of outgoing edges of the original IOR-split. It creates a linear number of additional sinks. Note that multiple sinks can be merged into a single sink by using an IOR-join. The IOR-join can then in turn be replaced by XOR and AND logic as we describe later in Sect. 4.4 and Sect. 4.5. Note that such a replacement is not always local, i.e., it does not necessarily fully preserve the structure, (cf. Sect. 4.5).

4.4. Local replacements for IOR-joins

Fig. 4.3 illustrates an example of a *local replacement* of an IOR-join. In this example, the IOR-join j is replaced by the partial workflow graph composed of the nodes v and w, and the edge i. As a graphical convention, we represent the elements of the original workflow graph using solid lines and the elements introduced to replace an

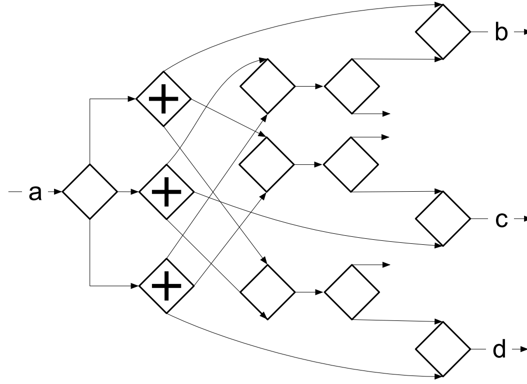
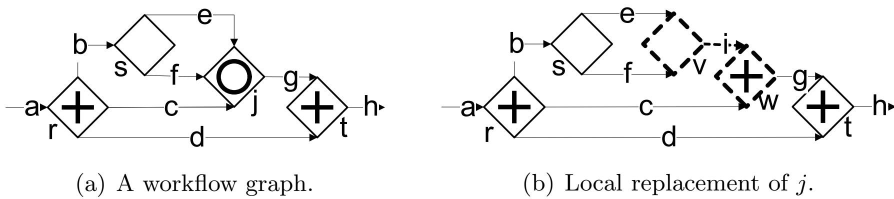


Figure 4.2.: A faithful quadratic translation of an IOR-split creating multiple sinks.

IOR-join using dashed lines. In the following, we omit tasks in the workflow graphs when they are not relevant for our discussion.



(a) A workflow graph.

(b) Local replacement of j.

Figure 4.3.: An example of local replacement where the IOR-join j is replaced by the partial workflow graph composed of the nodes v and w, and the edge i.

Intuitively, we observe that the behavior of the local replacement and the IOR-join are equivalent: The execution of the IOR-join in Fig. 4.3(a) is enabled if and only if either the edges e and c are marked or the edges f and c are marked. The execution of the local replacement in Fig. 4.3(a) is enabled under the exact same conditions. In the following, we formalize the definition of local replacement and the equivalence between the original workflow graph and the workflow graph obtained by local replacement.

4.4.1. Local replacement and equivalence

A local replacement of an IOR-join j is a partial workflow graph R that connects to the edges left dangling by the removal of j. Note that j and R have exactly the same set of incoming and outgoing edges. Apart from the IOR-join, a local replacement preserves the original workflow graph, which makes it very easy to relate the original and the translated workflow graph. We formalize a local replacement as follows:

Definition 4.1 (Local replacement). *Let $W = (N, E, c, l)$ be a workflow graph and j be a node in N. Let $R = (N'', E'', c'', l'')$ be a partial workflow graph such that for each*

node $n \in N''$, $l''(n) = \text{XOR}$ *or* $l''(n) = \text{AND}$, $N \cap N'' = \emptyset$, *and* $E \cap E'' = \emptyset$.

A local replacement *of* j *in* W *by* R *results in a workflow graph* $W' = (N', E', c', l')$ *such that:*

- $N' = N \setminus \{j\} \cup N''$,
- $E' = E \cup E''$,
- $c'(e) = c''(e)$ *when* $e \in E''$,

  $c'(e) = c(e) = (s, t)$ *when* $e \in E$ *and* $s \neq j \neq t$,

  $c'(e) = (s, t)$ *such that* $s \in N''$ *and* $t \in N$ *if* $e \in E$ *and* $c(e) = (j, t)$,

  $c'(e) = (s, t)$ *such that* $t \in N''$ *and* $s \in N$ *if* $e \in E$ *and* $c(e) = (s, j)$,
- $l'(n) = l(n)$ *when* $n \in N \setminus j$, $l'(n) = l''(n)$ *when* $n \in N''$, *and*
- *each element* $x \in N'' \cup E''$ *is on a path in* W' *from an edge* $e_{in} \in E$ *such that* $c(e_{in}) = (s, j)$ *to the edge* $e_{out} \in E$ *such that* $c(e_{out}) = (j, t)$.

Intuitively, the workflow graph W' resulting from the local replacement of an IOR-join j in a workflow graph W by a partial workflow graph R is *equivalent* to W if R has the same "behavior" as j. We will now formalize this intuition.

Let, in the rest of this section, $W = (N, E, c, l)$ be a workflow graph containing an IOR-join j and $W' = (N', E', c', l')$ be a workflow graph obtained by local-replacement of j by a partial workflow graph R that does not contain any IOR-join. We first introduce the notions of replacement transition and replacement execution sequence:

Definition 4.2 (Replacement transition and replacement sequence). *A transition* (E_1, n, E_2) *of* W' *is a* replacement transition *if* $n \in (N' \setminus N)$. *An execution sequence* σ *of* W' *is a* replacement execution sequence *if each transition of* σ *is a replacement transition and after* σ *no replacement transition is enabled and no edge* $e \in E' \setminus E$ *is marked.*

We can now define a notion of equivalence between W and W':

Definition 4.3 (Equivalence of a local replacement). W *and* W' *are* equivalent *if the following two conditions are met:*

1. *Let* m_1 *and* m_2 *be two reachable markings of* W. *For any transition* $t = (E_1, j, E_2)$ *such that* $m_1 \xrightarrow{t} m_2$ *in* W, *there exists a replacement execution sequence* σ *such that* $m_1 \xrightarrow{\sigma} m_2$ *in* W'.
2. *Let* m_1 *and* m_2 *be two reachable markings of* W' *such that* m_1 *and* m_2 *only mark edges in* E. *For any replacement execution sequence* σ *such that* $m_1 \xrightarrow{\sigma} m_2$ *in* W', *there exists a transition* $t = (E_1, j, E_2)$ *such that* $m_1 \xrightarrow{t} m_2$ *in* W.

4.4.2. Characterization of locally replaceable IOR-joins and replacement technique

In the following, we give a local replacement technique which, as we will see later, can provide a local replacement for any IOR-join that can be replaced locally. Then, we characterize under which conditions an IOR-join can be replaced locally, i.e., regardless of the replacement technique. This result is an extension of a technique (76) that completes a workflow graph with multiple sinks to obtain a workflow graph with a single sink.

Some IOR-joins can easily be replaced locally: It is clear that we can replace an IOR-join by an AND-join if all its incoming edges are marked every time it is executed, and by an XOR-join if only one of its incoming edge is marked every time it is executed (83). For an acyclic workflow graph, we have shown elsewhere (26) how to compute these properties in quadratic time with respect to the size of the workflow graph. Furthermore, a workflow graph completion heuristic based on the *refined process structure tree* (76) can also provide a local replacement for some IOR-joins.

The idea behind the local replacement that we propose is to merge groups of edges which are never marked together with an XOR-join using enough edges to ensure that an edge of the group is marked during any execution where the IOR-join is executed. Then we join the outgoing edges of the created XOR-joins by an AND-join. This strategy also supports the simple cases described earlier. After defining this strategy, we will present examples of this replacement and discuss the limitations of local replacements in Sect. 4.4.3.

First, we define the notions of test and cover which will allow us to formulate the replacement based on covers:

Definition 4.4 (Mutually exclusive edges, test, and cover)**.** *Let $W = (N, E, c, l)$ be a workflow graph.*

Two edges in E are mutually exclusive *if there exists no execution of W which marks both edges.*

A test *of an edge $e \in E$ is a set $T_e \subseteq E$ of pairwise mutually exclusive edges such that an execution σ of W marks e if σ marks one of the edges in T_e.*

Let $X \subseteq E$ and $e \in E$. A cover *of X* with respect to *e is a set $\mathcal{C}$ of tests of e such that for each $T_e \in \mathcal{C}$ $T_e \subseteq X$ and each edge in X belongs to a test in $\mathcal{C}$.*

Note this definition allows a test to contain a single edge. In Fig. 4.3(a), the tests $T_1 = \{e, f\}$ and $T_2 = \{c\}$ of g form a cover $\mathcal{C} = \{T_1, T_2\}$ of $\circ j$ with respect to g. We now describe how to obtain a local replacement of an IOR-join using a cover of its incoming edges with respect to its outgoing edge. We shall see later that such a cover does not always exist. Fig. 4.4 illustrates the structure of the replacement:

Definition 4.5 (Cover-based replacement)**.** *Let j be an* IOR*-join in a workflow graph W and o be the outgoing edge of j. Let $\mathcal{C}$ be a cover of $\circ j$ with respect to o.*

Let the $R = (N'', E'', c'', l'')$ be a partial workflow graph defined as follows:

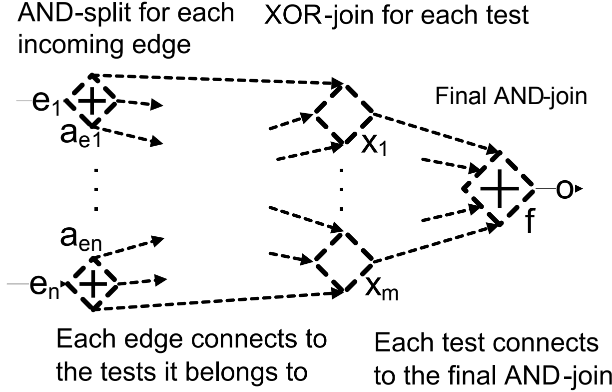


Figure 4.4.: Canvas of cover-based replacement.

- *N'' contains: an* AND*-split a_e for each edge $e \in \circ j$, an* XOR*-join x_i for each test $T_i \in \mathcal{C}$, and one* AND*-join f.*

- *For each test $T_i \in \mathcal{C}$, for each edge $e \in T_i$, E'' contains an edge from the* AND*-split a_e to the* XOR*-join x_i. For each* XOR*-join x_i, E'' contains an edge from x_i to the* AND*-join f.*

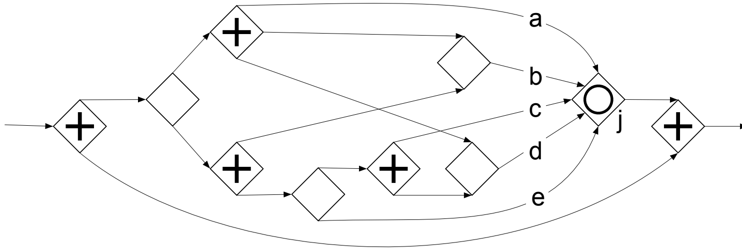
The Cover-based replacement *(*C-replacement *for short) of j replaces j by R as follows: The target of each (previously) incoming edge e of j is set to be a_e. The source of the (previously) outgoing edge o of j is set to f.*

Note that when an edge e in $\circ j$ belongs to only one test, the AND-split a_e has a single outgoing edge and can be removed. When a test T_i contains only one edge, the XOR-join x_i has a single incoming edge and can be removed.

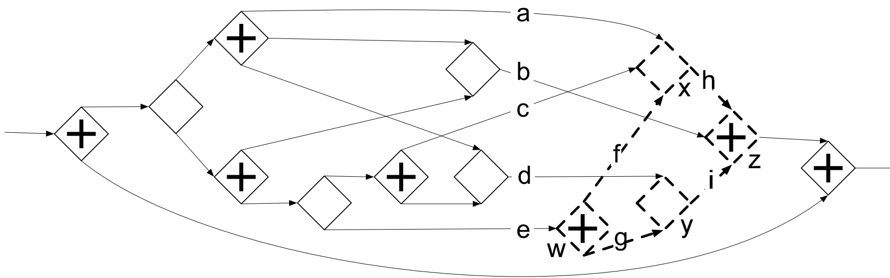
The local replacement illustrated by Fig. 4.3(a) and Fig. 4.3(b) is the result of a C-replacement using the cover $\mathcal{C} = \{\{e, f\}, \{c\}\}$ of $\circ j$ with respect to the outgoing edge g of j. Because each edge is used only in one test, there is no AND-split necessary. Moreover, the test $\{c\}$ contains only one edge, therefore it does not require an XOR-join.

Fig. 4.5 illustrates a more complex C-replacement of the IOR-join labeled j of Fig. 4.5(a) by the partial workflow graph containing the nodes w, x, y, and z and the edges f, g, h, and i in Fig. 4.5(b). This replacement is based on the cover $\mathcal{C} = \{\{a, c, e\}, \{b\}, \{d, e\}\}$.

Computing tests, including checking that edges are mutually exclusive can be done using state space exploration, which can take exponential time. More efficient heuristics exist for some special cases. For example, it is possible to compute in quadratic time whether a set of edges in an acyclic process is mutually exclusive (26). Efficient computation of the tests is out of scope of this thesis.



(a) A workflow graph.



(b) Local replacement of j.

Figure 4.5.: An example of local replacement where the IOR-join j is replaced by the partial workflow graph composed of the nodes w, x, y, and z and the edges f, g, h, and i.

We first prove the following lemma which states that an execution can follow a path. This lemma will help us in various proof including characterizing when an IOR-join is locally replaceable:

Lemma 4.6. *Let W be a deadlock-free workflow graph. Let e, e^*, e' be three edges of W such that there exists no path from e^* to e'. Let m be a reachable marking of W which marks e and e^*. Let p be a path from e to e'.*

There exists a marking m', reachable from m, such that m' marks e' and e^*.

Proof. We proceed by induction on the length of p.

Base case: length(p) = 1 (i.e., $e = e'$). The marking m marks e^* and e'.

Induction step: Assuming that Lemma 4.6 holds for any path of length n, we show that it holds for any path of length $n+1$. Let e_n be the nth edge of p, e' be the next edge, and n be the node between e_n and e'. The prefix of p until e_n is a path of length n from e to e_n. Moreover, there is no path from e^* to e_n otherwise

there would be a path from e^* to e'. Therefore, by induction assumption, there exists a marking m_n, reachable from m, such that m_n marks e_n and e^*. By the workflow graph semantics, if n is an XOR-join, an XOR-split, an AND-split, or an IOR-split, there exists a transition t such that $m_n \xrightarrow{t} m'$ and m' marks e_* and e'. We are left with the cases where n is an IOR-join or an AND-join. Any execution sequence eventually executes the target of e_n, because otherwise there would be a deadlock, which would contradict the assumption that the workflow graph is deadlock-free. Therefore, by the workflow graph semantics (Def. 2.18), e' eventually gets marked by any execution. We are left to show that there exists an execution sequence which does not move the token on e^*. We do so by showing that target of e^*. If we suppose that the execution of the target n of e^* is required to mark e' then there would be path from n, and therefore from e^*, to e' which would contradict the initial assumption of the lemma.

□

We can now characterize the conditions under which an IOR-join has an equivalent local replacement:

Theorem 4.7 (Equivalent local replacement existence)**.** *Let W be a sound workflow graph containing an* IOR*-join j.*

An IOR*-join j has an equivalent local replacement iff there exists a cover of $\circ j$ with respect to the outgoing edge of j.*

Proof. We show the two directions of the theorem separately:

$\Leftarrow$ Let C be a cover of $\circ j$ with respect to the outgoing edge of j. We show that there exists a local replacement by showing that the C-replacement R of j results in a workflow graph W' that is equivalent to W. We use the usual labeling for the nodes of R as illustrated by Fig. 4.4. Let m_1, m_2 be two markings of W (and thus two markings of W' because $E \subseteq E'$):

1. For any transition $t = (E_1, j, E_2)$ such that $m_1 \xrightarrow{t} m_2$ in W, there exists a replacement execution sequence σ such that $m_1 \xrightarrow{\sigma} m_2$ in W':

   By the IOR-join semantics, we have $m_2 = m_1 - E_1 + e_o$ and $E_1 \neq \emptyset$. Note that, because j is enabled in m_1 and W is sound, there is no token upstream of an edge in $\circ j$. We now build the execution σ of R that changes m_1 to m_i by executing the AND-splits a_e that have a marked incoming edge and then the XOR-joins of R. By the test definition and the definition of C-replacement, each XOR-join has one marked incoming edge and, by the property of mutual exclusion of the edges in a test, only one of incoming edge is marked. Then the final and join f of R is executed. The AND-join f is enabled in m_i because each XOR-join was executed and thus, by

C-replacement definition and XOR-join semantics, each incoming edge of f carries a token. Executing f changes m_i into m_2.

2. For any replacement execution sequence σ such that $m_1 \xrightarrow{\sigma} m_2$ in W', there exists a transition $t = (E_1, j, E_2)$ such that $m_1 \xrightarrow{t} m_2$ in W: We must show that:

   a) t is enabled in m_1: Suppose that t is not enabled in m_1. By IOR-join semantics either no incoming edge of j is marked in m_1 or there is a marked edge preceding a non-marked incoming edge of j. Because σ is a replacement transition sequence, all transitions of σ execute a node in R. Thus, because m_1 only marks edges in E and by the definition of local replacement, an edge of $\circ j$ is marked. Thus, there must be an edge e that precedes an edge $e' \in \circ j$ which does not carry a token in m_1. By the cover definition, e' belongs to a test T, and by C-replacement and the definition of replacement execution sequence, there exists an edge e'', such that $e'' \neq e'$, which belongs to T and is marked by m_1. As W is deadlock-free by assumption, there is a path from e to e', and e'' is not on a path to or from e', by Lemma 4.6, there exists an execution sequence σ' and a marking m_1' such that $m_1 \xrightarrow{\sigma'} m_1'$ and the edges e', e'' carry a token in m_1'. This is in contradiction with the definition of a test as all the edges in a test are mutually exclusive.

   b) executing t results in m_2: like t, σ consumes one token of each incoming edge marked in m_1 because if an edge of $\circ j$ was marked in m_2, then a replacement transition would be enabled which would contradict the definition of replacement execution sequence. It is also clear that, like executing σ, executing t marks the (formerly) outgoing edge of j by the IOR-join semantics.

$\Rightarrow$ We show that when an IOR-join j in a workflow graph $W = (N, E, c, l)$ can be replaced locally by a partial workflow graph R (not necessarily using a C-replacement) to result in an equivalent workflow graph $W' = (N', E', c', l')$, then there exists a cover of $\circ j$ with respect to the outgoing edge e of j.

A set X of edges is an *independent set* iff for each pair of edge $e_1, e_2 \in X$ there exists no path from e_1 to e_2. An independent set X_1 is *maximal with respect to* a graph G iff there exist no independent set X_2 of edge in G such that $X_1 \subset X_2$. Let $G = R \cup \circ j \cup \{e\}$. In the following, whenever we mention a maximum independent set we omit to indicate that it is with respect to G. It is clear that $\{e\}$ and $\circ j$ are both maximum independent sets. Let δ be a function which for each ordered pair of maximum independent sets (X_1, X_2) returns the number of edges, not included in $X_1 \cup X_2$ on a path from an edge in X_1 to an edge in X_2.

We proceed by structural induction on G, starting from e and following the edges

backward. We show that at each step, given a maximum independent set X s.t. $X \neq \circ j$ and a cover C of X with respect to e, we can build a maximal independent set X' such that $\delta(\circ j, X') < \delta(\circ j, X)$ and a cover C' of X' with respect to e, which shows that there is a cover C^* of $\circ j$ with respect to e in W'. By definition of equivalence of the local replacement, it is clear that a test $T \subseteq E$ of an edge $e \in E$ in W' is also a test of e in W. Thus C^* is a cover of $\circ j$ with respect to the outgoing edge of e in W.

Base case: $\{\{e\}\}$ is a cover of $\{e\}$ with respect to e.

Induction step: Assume C to be a cover of $X \subseteq G$ with respect to e such that X is a maximum independent set. We show that for each edge u with the node n as target such that $n\circ \subseteq X$, there exists a cover C' of a set $X' \subseteq G$ with respect to $\{e\}$ such that $u \in X'$, X' is a maximum independent set, and $\delta(\circ j, X') < \delta(\circ j, X)$. We distinguish three cases:

1. n is an XOR-join: Without loss of generality, we can assume that n has two incoming edges u, v and one outgoing edge w. Let $X' = X \cup \{u, v\} \setminus \{w\}$. Because X is a maximum independent set, it is clear that X' is a maximum independent set such that $\delta(\circ j, X') < \delta(\circ j, X)$. We transform C into a cover C' of X' with respect to e by replacing w by u and v in all test of C.

   We first show that if a test $T \in C$ of e contains w, then $T' = T \setminus \{w\} \cup \{u, v\}$ is a test of e:

   We show that the edges in T' are pairwise mutually exclusive: Because T is a test, all edges in $T \setminus \{w\}$ are mutually exclusive. By the soundness assumption of W, the edges u and v are mutually exclusive. Moreover, the edges in $T \setminus \{w\}$ are pairwise mutually exclusive with u and v because otherwise they would not be mutually exclusive with w by the XOR-join semantics. Thus, all edges of T' are pairwise mutually exclusive.

   To prove that T' is a test, we are left to show that, for any execution σ, σ marks e iff σ marks an edge of T': from the construction of T' and the XOR-join semantics which ensures that σ marks w iff σ marks an edge in $\circ n$, we have that σ marks an edge in T' iff σ marks an edge in T. Because T is a test, it implies that σ marks T' iff σ marks e.

2. n is an XOR-split: Without loss of generality, we can assume that n has two outgoing edges v, w.

   We first show that a test $T \in C$ containing v must contain w (and vice et versa): Suppose that a test $T \in C$ contains v but not w. Let σ be an execution such that σ marks w. By the test definition, there exist an edge $x \in T$ that is marked by σ. Note that, by definition of C, we also have $x \in X$. Let σ' be a prefix of σ such that the last marking m_1 of σ' is

followed in σ by m_2 and m_2 is the first marking in σ which marks w or x. We can assume, without loss of generality, that m_2 marks w and thus m_1 marks the incoming edge u of v. Because x is marked in a state following m_1 during σ, there exists an edge x' and a path p from x' to x such that m_1 marks x'. As X is a maximum independent set, there exists no path between x and w, which implies that there exists no path between u and x.ar By Lemma 4.6, there exists a marking m' reachable from m_1 such that m' marks u and x. The transition $t = (\{u\}, n, \{v\})$ is enabled in m' by XOR-split semantics and because m' marks u. Executing t in m' results in a marking m'' such that m'' marks v and x. As there exists a marking which marks v and x, v and x are not mutually exclusive, so T is not test. We have shown that a test containing v must contain w. (The same reasoning can be applied to show that a test containing w must contain v.)

Let $X' = X \cup \{u\} \setminus \{v, w\}$. Because X is a maximum independent set, it is clear that X' is a maximum independent set such that $\delta(\circ j, X') < \delta(\circ j, X)$. We transform C into a cover C' of X' with respect to e by replacing v and w by u in all test of C. Using a similar reasoning as done the previous case n is an XOR-join, it can be shown that if a test $T \in C$ of e contains v and w, then $T' = T \setminus \{v, w\} \cup \{u\}$ is a test of e.

3. $l(n) = AND$: Let $X' = X \cup \circ n \setminus j\circ$. Because X is a maximum independent set, it is clear that X' is a maximum independent set such that $\delta(\circ j, X') < \delta(\circ j, X)$. We transform C into a cover C' of X' with respect to e by as follows: For each test $T \in C$ such that T contains an edge $v \in j\circ$, we create a test T_u for each edge $u \in \circ j$ such that $T_u = T \cup \{u\} \setminus \{v\}$. We replace T in C by the tests $\bigcup_{u \in \circ j} T_u$ to obtain C'. It is easy to see that each T_u is a test of e and that C' is a cover of X' with respect to e.

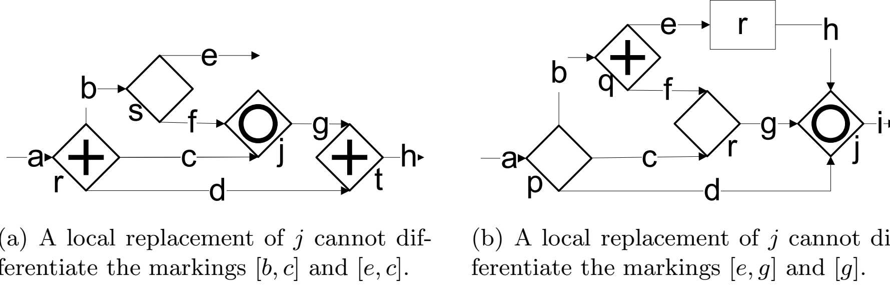
□

While Thm. 4.7 applies to any local replacement technique, the proof of the 'if' direction shows that, whenever there exists a cover of $\circ j$ with respect to the outgoing edge of j, the C-replacement of j produces an equivalent workflow graph.

4.4.3. The limitation of local replacements

The local replacement replaces an IOR-join by gateways which have local semantics and, as the replacement retains the same incoming edges as the IOR-join, only receives local "information". In some cases, this local information is not enough to differentiate between some reachable markings that can be differentiated by the non-local semantics

of the IOR-join. The two workflow graphs illustrated by Fig. 4.6 are workflow graphs that cannot be locally replaced for this reason. The 'only if' direction of Thm. 4.7 allows us to capture this limitation.



(a) A local replacement of j cannot differentiate the markings $[b, c]$ and $[e, c]$.

(b) A local replacement of j cannot differentiate the markings $[e, g]$ and $[g]$.

Figure 4.6.: Two workflow graphs containing an IOR-join that cannot be replaced locally.

Fig. 4.6(a) is a slight variation of Fig. 4.3(a) where changing the target of the edge e makes it impossible replace the IOR-join locally. By changing the target of e, e does not belong to $\circ j$ anymore. As in Fig. 4.3(a), the marking $m_1 = [b, c]$ does not enable j because there is a path from the token on b to the empty slot f and the marking $m_2 = [e, c]$ enables j. In contrast to Fig. 4.3(a), in Fig. 4.6(a), the markings m_1 and m_2 cannot be differentiated based only on the local "information", i.e., based on the marking of the edges f and c. In terms of Thm. 4.7, the test $T_1 = \{e, f\}$ cannot be used to build a cover of $\circ j$ anymore and there is no other test of g that contains f.

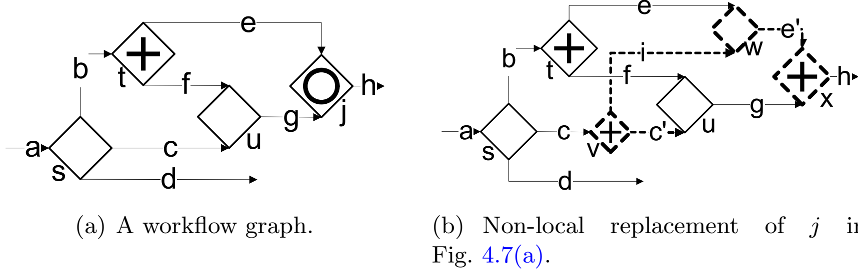
In Fig. 4.6(b), a local replacement would not be able the to differentiate the two reachable marking $[e, g]$ and $[g]$. In terms of Thm. 4.7, the IOR-join cannot be replaced locally because e does not belong to any test of h contained in $\circ j$, i.e., there exists no cover of h.

We will see in the next section how the IOR-joins of these two examples can be replaced using a non-local replacement which allows us to transfer the "missing information".

4.5. Non-local replacements for IOR-joins

Fig. 4.7(b) shows an example of a non-local replacement where the IOR-join j of Fig. 4.7(a) is replaced by the partial workflow graph composed of the nodes w, x and the edges i, e' where, additionally, the AND-split v was inserted on the edge c which delivers additional (non-local) information to the IOR-join replacement via the edge i.

So, in addition to a local replacement, we allow non-local replacements to insert additional AND-splits in the graph, which can only be connected to the IOR-join replacement. These AND-splits only "copy" tokens to route them to the replacement



(a) A workflow graph.

(b) Non-local replacement of j in Fig. 4.7(a).

Figure 4.7.: An example of non-local replacement.

and do not alter the original behavior of the process. This also preserves the graph structure to a large extent.

Kiepuszewski et al. (42, Proof of Theorem 5.1) give a completion approach to transform a Petri net with multiple sinks into a Petri net with a single sink. In this section, we will show that one can use a variation of that approach, which we call *K-replacement*, to replace an IOR-join in an acyclic workflow graph. We give a condition that characterizes the IOR-joins for which this replacement produces an equivalent workflow graph. Finally, we show that checking whether replacing the IOR-join produces an equivalent workflow graph can be done in polynomial time and that the replacement itself requires polynomial time.

4.5.1. Non-local replacement and equivalence

We now formalize the concept of non-local replacement introduced in the previous section. When *inserting* a gateway g on an edge e, we create an additional edge e' such that the source of e' is g and the target of e' is the target of e and the target of e becomes g. We say that the edge e' is the *resulting edge* from the insertion of g on e.

Definition 4.8 (Non-local replacement)**.** *Let* $W = (N, E, c, l)$ *be a workflow graph and* j *be a node in* N*. Let* $R = (N'', E'', c'', l'')$ *be a partial workflow graph such that for each node* $n \in N''$, $l''(n) = \text{XOR}$ *or* $l''(n) = \text{AND}$, $N \cap N'' = \emptyset$, *and* $E \cap E'' = \emptyset$*. Let* $l_g =< a_0, \ldots, a_n >$ *be a list of* AND*-splits such that* $l_g \cap (N \cup R) = \emptyset$.

A non-local replacement *of* j *in* W by R *and* l_g *results in a workflow graph* $W' = (N', E', c', l')$ *obtained by the* insertion *of* l_g *on some edges* $< e_0, \ldots, e_n >$ *of* W *resulting in a list* l_e *of edges and the* insertion *of* R *such that:*

- $N' = N \setminus \{j\} \cup N'' \cup l_g$,
- $E' = E \cup E'' \cup l_e$,
- $c'(e) = c(e) = (s, t)$ *when* $e \in E$, $s \in N$, $t \in N$, *and* $s \neq j \neq t$,

  $c'(e) = c''(e) = (s, t)$ *when* $e \in E''$, $s \in N''$, *and* $t \in N''$,

$c'(e) = (s, t)$ such that $s \in N$ and $t \in N''$ if $e \in E$ and $c(e) = (s, j)$ or $e \in E''$ and $s \in l_g$,

$c'(e) = (s, t)$ such that $s \in N''$ and $t \in N$ if $e \in E$ and $c(e) = (j, t)$,

- *$l'(n) = l(n)$ when $n \in N \setminus j$, $l'(n) = l''(n)$ when $n \in N''$, and*
- *each element $x \in N'' \cup E''$ is on a path in W from an edge $e_{in} \in E$ such that $c(e_{in}) = (s, j)$ or a node $a_i \in l_g$ to the edge $e_{out} \in E$ such that $c(e_{out}) = (j, t)$.*

Relating the definition of non-local replacement to the example discussed earlier and illustrated by Fig. 4.7(a) and Fig. 4.9, we have that W is illustrated by Fig. 4.7(a) and W' is illustrated by Fig. 4.9. The elements of the replacement are illustrated by dashed lines. The list l_g of inserted AND-splits contains only v and the edge resulting from the insertion of v is c'.

We now define when a non-local replacement is semantically correct through a notion of equivalence of the two workflow graphs. Let, in the rest of this section, $W = (N, E, c, l)$ be a workflow graph containing an IOR-join j and $W' = (N', E', c', l')$ be a workflow graph obtained by non-local replacement of j. To define equivalence we need to map the markings of W and W'. We first define a mapping $\psi : E' \to E \cup \{null\}$ such that for any edge e' in E':

$$\psi(e') = \begin{cases} e' & \text{if } e' \in E, \\ e & \text{if } e' \text{ is the resulting edge of the insertion of an AND-join on } e, \\ null & \text{otherwise.} \end{cases}$$

We define a mapping ϕ from a marking of W' to a marking of W such that $\phi(m)[e] = \sum_{\psi(e')=e} m[e']$.

We reuse the notion of *replacement execution sequence* defined in Sect. 4.4.

Definition 4.9 (Equivalence of non-local replacement)**.** *W and W' are* equivalent *if for any pair of reachable markings m_1 of W, m'_1 of W' such that $m_1 = \phi(m'_1)$, we have:*

1. *for any transition $t = (E_1, j, E_2)$ and any marking m_2 such that $m_1 \xrightarrow{t} m_2$ in W, there exists a replacement execution sequence σ and a marking m'_2 such that $m'_1 \xrightarrow{\sigma} m'_2$ in W' and $m_2 = \phi(m'_2)$, and*
2. *for any replacement execution sequence σ and any marking m'_2 such that $m'_1 \xrightarrow{\sigma} m'_2$ in W', there exists a transition $t = (E_1, j, E_2)$ and a marking m_2 such that $m_1 \xrightarrow{t} m_2$ in W and $m_2 = \phi(m'_2)$.*

4.5.2. A simple non-local replacement

We now discuss informally a simple version of a non-local replacement technique, which we call *simple K-replacement*. We will then point out a shortcoming of simple K-replacement and modify it to obtain the *K-replacement*. The description of the simple K-replacement is an intermediary step towards the description of the more complex K-replacement.

Fig. 4.8(a) and Fig. 4.8(b) show equivalent non-local replacements for the workflow graphs in Fig. 4.6(a) and Fig. 4.6(b), respectively. These are two examples of simple K-replacement.

(a) Non-local replacement of j in Fig. 4.6(a).

(b) Non-local replacement of j in Fig. 4.6(b).

Figure 4.8.: Non-local replacements of the IOR-joins in Fig. 4.6 that do not have a local replacement.

The simple K-replacement uses the notion of a *bridge*: A *bridge* from an edge e to an edge e' is a path from e to e' such that each split on the path is an AND-split and each join on the path is an XOR-join. For example, in Fig. 4.8(a), the path $< e, v, j, w, f' >$ is a bridge from e to f'. The existence of a bridge from e to e' implies that every execution that marks e also marks e'.

The key idea of simple K-replacement is to replace an IOR-join by an AND-join and to ensure that every incoming edge of the new AND-join carries a token in every execution by adding suitable bridges. More precisely, for each incoming edge e' of the AND-join and each outgoing edge e of any XOR-split such that e is not on a path to e', we create a bridge from e to e'. Intuitively, for each outgoing edge e of an XOR-split that removes a token from a path to e', we add a bridge that brings an additional token to e' on a different path. This leads to equivalent workflow graphs for the examples in Fig. 4.6(a) and Fig. 4.6(b).

However, consider the workflow graph in Fig. 4.9 obtained by the same technique for the IOR-join v in the workflow graph shown by Fig. 4.7(a). When an execution σ marks the edge d of the workflow graph in Fig. 4.9, the edge h is also marked by σ which is not the case when an execution marks the edge d in the workflow graph in Fig. 4.7(a).

In summary, the simple K-replacement leads to an equivalent workflow graph if and only if it is applied to an IOR-join that is executed by every execution of the original workflow graph.

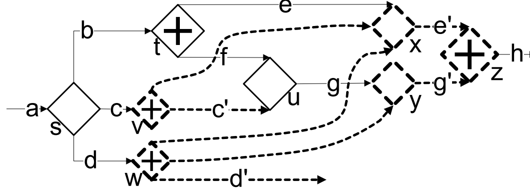


Figure 4.9.: Non-local replacement of j in Fig. 4.7(a).

4.5.3. K-replacement

We now describe our generalized technique, called *K-replacement*. The K-replacement subsumes the simple K-replacement that we described earlier. Applying K-replacement to f in Fig. 4.7(a) results in the workflow graph in Fig. 4.7(b). K-replacement uses the notion of *dominator frontier* to apply the same replacement strategy as the simple K-replacement to a sub-graph of the workflow instead of the complete graph. Applying the K-replacement to a sub-graph of the workflow graph implies that the AND-join replacing the IOR-join only executes during the execution that marks an edge of this sub-graph and thus allows us to produce an equivalent replacement in more cases than with simple K-replacement.

To this end, we use the notions of dominator and dominator frontier.

Definition 4.10 (Dominator and dominator frontier)**.** *A node x_1* dominates *another node x_2 if each path from the source edge of the workflow graph to x_2 contains x_1.*

A dominator x_1 of a node x_2 is the minimal dominator *of x_2 if there exists no node x_1' such that x_1' dominates x_2, x_1 dominates x_1', and $x_1 \neq x_1' \neq x_2$.*

A set E_d of edges is the dominator frontier *of a node x_2 if a node x_1 is the minimal dominator of x_2, $E_d \subseteq x_1\circ$, for each edge $e \in E_d$, $e < x_2$, and for each edge $e' \in ((x_1\circ) \setminus E_d)$, we have $e' \not< x_2$.*

For example, in Fig. 4.3(b), the nodes r and s dominate the node v and the node s is the minimal dominator of v. In Fig. 4.7(a), the dominator frontier of j is the set of edges $\{b, c\}$.

K-replacement replaces an IOR-join j by an AND-join. Furthermore, a bridge from e to e' is created for each incoming edge e' of j and each edge e such that e is the outgoing edge of an XOR-split on a path from an edge of the dominator frontier of j to j, and there is no path from e to e'. K-replacement is detailed further by Algorithm 5. As mentioned earlier, Fig. 4.7(b) shows the workflow graph resulting from the application of Alg. 5 to the IOR-join j in Fig. 4.7(a).

The K-replacement implies that the AND-join replacing the IOR-join executes in every execution where an edge of the dominator frontier is marked. Thus, intuitively, the K-replacement produces an equivalent workflow graph if each execution that marks one edge of the dominator frontier also executes the IOR-join.

Algorithm 5 K-replace(j, W).
Input: A workflow graph $W = (N, E, c, l)$ and IOR-join $j \in N$.
Output: A workflow graph $W' = (N', E', c', l')$ where j has been K-replaced.

```
Create an AND-join a and set the source of the outgoing edge of j to be a.
Let E_I be the set of incoming edges of j in W.
for each edge e ∈ E_i do
    Create an XOR-join x_e.
    Set the target of e to be x_e.
    Create and edge from x_e to a.
end for
Let preset(j) be the set of all elements in E ∪ N from the minimal dominator of j
having a path to j.
for each edge e' ∈ ∘j do
    for each decision d ∈ preset(j) do
        Let preset(e') be the set of all elements in E ∪ N from the dominator frontier
        of j having a path to e'.
        for each edge e ∈ d∘ such that e ∉ preset(e')) do
            if d ≠ minimal dominator of j OR e ∈ dominator frontier of j then
                if The target of e is not an and split then
                    Insert an AND-split s on e
                else
                    Let s be the target of e
                end if
                Add an edge from s to x_e'
            end if
        end for
    end for
end for
```

In the following, we formalize this intuition as a necessary and sufficient condition for the K-replacement to produce an equivalent workflow graph. We start by proving Lemma 4.11 which will be useful to prove the condition of applicability given by Thm. 4.12:

Lemma 4.11. *Let W' be the workflow graph obtained by K-replacement of j. Let f be the minimal dominator of j in W. Let e' be an edge in $\circ j$ in W. Let x be the XOR-join targeted by e' in W' and e'' be the outgoing edge of x.*

For any execution σ of W' that executes f, there exists a path p in W' from f to e'' such that each edge of p is marked during σ.

Proof. By the definition of dominator it is clear that a path from f to e' exists in W and thus that a path p_1 from f to e'' exists in W'.

Either each edge of p is marked during σ, in which case we are done, or there exists an edge of p that is not marked during σ. Let e^* be the first edge of p that is not marked during σ. As e^* is the first edge of p that is not marked during σ and f is executed during σ, the node n preceding e^* must be executed during σ. The node n must be an XOR-split, otherwise the marking resulting from its execution would contradict the workflow graph semantics. Moreover, one of the outgoing edges e_2 of n is marked after execution of n. If e_2 is not on a path to e', then by the K-replacement procedure, the target of e_2 is an AND-split with a path p^* to e'' that is be marked by σ. The concatenation of the prefix of p_1 until n and p_* is a path p from f to e'' such that all edges of p are marked during σ. If p is on a path p_2 to e', we can repeat with p_2 the reasoning we followed with p_1. As we are in an acyclic graph the length of any path in W' is bounded by the number of edges in W' and therefore this reasoning terminates. □

We can now prove that an IOR-join j has an equivalent K-replacement iff j is executed in each execution where an edge of the dominator frontier of j carries a token:

Theorem 4.12 (K-replacement applicability). *K-replacement of an* IOR*-join j in a sound workflow graph W produces a workflow graph that is equivalent to W iff j is executed in each execution of W in which an edge of the dominator frontier of j is marked.*

Proof. We show the two directions of the theorem separately:

⇒ We prove the contrapositive statement: Assume that there exists some execution σ such that dominator of j is executed in σ but j is not executed in σ.

  We show by contradiction that K-replacement does not lead to an equivalent workflow graph: Suppose that there exists a valid K-replacement of j resulting in a workflow graph W' such that $W \equiv W'$. Consider the first marking m_2 in σ such that there is no edge preceding j that is marked in m_2 and the previous marking m_1 in σ. It is clear that there exists an edge e_2 that is marked in m_2 and

not in m_1 such that e_2 does not precede j. The edge e_1 preceding e_2 is marked in m_1 and precedes j. By the completion definition (Alg. 5), in W', an AND-split a is inserted on e_2 to obtain W' and therefore there is a path from e_2 (which is the incoming edge of a) to the previously outgoing edge of j. There exists a marking m'_2 of W' such that $m_2 \equiv m'_2$ and e_2 is marked in m'_2. By Lemma 4.11 and because there is no deadlock by the assumption that W is sound, there exists an execution sequence in W' from m'_2 to a marking in which the (previously) outgoing edge of j carries a token. There exists no such execution sequence from m_2 in W. Therefore, we have $W \not\equiv W'$.

$\Leftarrow$ Assume that j is executed in each execution where an edge of the dominator frontier F of j carries a token.

Let $W' = (N', E', c', l')$ be the workflow graph obtained by K-replacement of j. Let m_1 be a reachable marking of W and m'_1 be a reachable marking of W' such that $m_1 = \phi(m'_1)$

We show that W and W' are equivalent:

1. Let $t = (E_1, j, E_2)$ be a transition executing j such that $m_1 \xrightarrow{t} m_2$ in W, we show that there exists a replacement execution sequence σ and a marking m'_2 such that $m'_1 \xrightarrow{\sigma} m'_2$ in W' and $m_2 = \phi(m'_2)$:

   Because j is enabled in m_1, either some XOR-joins of the replacement are enabled in m'_1, or some incoming edge of the AND-join a of the replacement are marked, or both. The execution σ starts by executing the XOR-joins enabled in m'_1.

   After executing the enabled XOR-join we end up in a state m'_a in which some edges in $\circ a$ are marked. We aim to reach a state such that each edge in $\circ a$ is marked: Let's consider an edge $e'' \in \circ a$ such that e'' is not marked in m'_a. By Lemma 4.11 there exists a path p in W' from the minimal dominator of j in W to e'' such that all edges of p are marked during any execution where an edge in F carries a token. Because at least one incoming edge of a carries a token in m_a, the minimal dominator of j was already executed during σ. As e'' is not marked and the minimal dominator of j was executed during σ, there is a an edge e^* on the path p. The edge e^* cannot belongs to E, otherwise j would not be enabled in m'_1 because there would be a path from a token to an empty incoming edge of j. There exists a path from e^* to e'' containing nodes whose execution only consume tokens marking edges of the replacement, i.e, edges in $E' \setminus E$. By Lemma 4.6, we can reach a making which marks e''.

   This approach can be repeated for all empty incoming edges of a allowing us to reach a state where each edge in $\circ a$ is marked and thus a is enabled. The last transition of σ executes a resulting in m'_2.

We have $m_2 \equiv m_2'$ because: σ consumes a token on all edges that were (in W) incoming to j. As mentioned earlier, other transitions only consume tokens marking edges of the replacement. Executing a produces a token on its outgoing edge, which was (in W) and consumes a token on each of its incoming edges.

2. Let σ^* be a replacement execution sequence such that $m_1' \xrightarrow{\sigma^*} m_2'$ in W', we show that there exists a transition $t = (E_1, j, E_2)$ and a marking m_2 such that $m_1 \xrightarrow{t} m_2$ in W and $m_2 = \phi(m_2')$:

   By definition of replacement execution we have that the (previously) outgoing edge of j is marked by m_2'. By definition of K-replacement, we have that for any execution σ' of W' containing m_2', σ' contains a reachable marking m_0', preceding m_1' in σ', such that m_0' marks and edge of F.

   We show that j executes after m_1 in W: Let σ be the execution sequence of W which contains the sequence of marking defined by applying the function ϕ to the markings of $exec'$ until reaching the marking m_1. It is easy to see that the σ contains a marking m_0 of W such that m_1 is reachable from m_0 and $m_0 = \phi(m_0')$. By assumption, as m_0 is marked by σ, the outgoing edge o of j is marked by any execution sequence containing m_0. The edge o is marked by any marking obtained by executing j in W and any marking obtained by executing the final AND-join f of R in W'. Thus j must be executed after m_0. By replacement execution sequence definition, the final AND-join f of R is executed during σ^*. The IOR-join j executes after reaching m_1 during σ because if it was executed earlier then there would be a marking in σ' marking preceding m_1' that marks o and f would execute before σ^* in W' and σ^* would execute f a second time which would contradict the soundness assumption.

   We show that a transition t executing j is enabled by m_1: There exists no edge preceding f that is marked in m_2, otherwise the AND-join f could execute a second time by Lemma 4.15 which would contradict the soundness assumption. Let $A_j = \circ j \cup j\circ$ be the set of edges adjacent to j in W. By definition of replacement execution sequence and of the function ϕ, $\phi(m_1') \setminus A_j = \phi(m_2') \setminus A_j$, i.e., the edges marked by m_2' which are not adjacent to j in W are marked in m_1'. Thus, no edge preceding an edge of A_j is marked by m_1'. By definition of ϕ, no edge preceding an edge of A_j is marked by m_1. As we have shown that j executes after m_1, there exists an edge e preceding j marked by m_1. As e does not precede an edge of A_j, we have $e \in \circ j$. Therefore a transition t executing j is enabled by m_1 because m_1 marks at least one edge in $\circ j$ and m_1 does not mark an edge preceding a non-marked edge of $\circ j$.

   Executing j in m_1 results in a marking m_2. We have shown previously how

a replacement execution results in a marking m'_2 such that $m_2 = \phi(m'_2)$.

□

Thus, K-replacement of the IOR-join j in Fig. 4.7(a) produces an equivalent workflow graph shown in Fig. 4.7(b) because j executes in an execution σ if and only if an edge of its dominator frontier $\{b, c\}$ is marked by σ. This is not the case for the workflow graph in Fig. 4.10. The dominator frontier of y in Fig. 4.10 is the set $\{b, c\}$. The edges b and c are marked by every execution. Thus, applying the K-replacement algorithm to y would thus produce a workflow graph in which the task z is executed during every execution. It is not the case for the workflow graph in Fig. 4.10 in which the execution where the edges g and d are marked does not execute z.

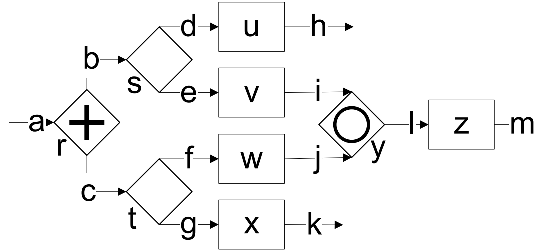


Figure 4.10.: The IOR-join y cannot be replaced.

4.5.4. A note on complexity

We now argue that the condition of applicability of the K-replacement, given by Thm. 4.12, can be computed efficiently, i.e., in polynomial time with respect to the size of the workflow graph. Alg. 5 also runs in polynomial time. This allows us to conclude that we can identify IOR-joins that can be replaced by K-replacement and replace them efficiently:

Theorem 4.13 (Polynomial time complexity of K-replacement)**.** *Let j be an IOR-join, and W be the workflow graph containing j.*

1. *Computing whether the K-replacement of j produces a workflow graph that is equivalent to W can be done in polynomial time with respect to the size of W.*
2. *The K-replacement of an* IOR*-join j can be computed in polynomial time with respect to the size of W.*

Proof. We show the polynomial complexity of the check and of the replacement separately:

1. In Chap. 5, we will present a *symbolic execution* for acyclic workflow graphs. The symbolic execution assigns to each edge of the workflow graph a symbol. These symbols allow us to check whether two edges e_1, e_2 of the workflow graph are marked by the exact same set of executions by computing whether the symbol assigned to e_1 and e_2 satisfy a certain notion of equivalence.

   To check the condition of applicability ok the K-replacement of an IOR-join j with denominator frontier D_f, we must check that, the outgoing edge of j and at the dominator frontier of j are marked by the same set of executions. We achieve this by checking that the symbol assigned to the outgoing edge of j is equivalent to the symbol obtained by summing the symbols assigned to the edges of D_f. The symbolic execution, summing the symbols assigned to the edges in D_f, and computing the equivalence, requires a quadratic amount of time with respect to the size of the workflow graph.

2. The domination relationship can be expressed using a tree data structure. Computing the dominator tree of W can be done in $O(|E|log(|N|))$ (46). Checking if an outgoing edge of the minimal dominator is part of the dominator frontier is a simple reachability test which requires linear time. Applying Alg. 5 clearly requires a polynomial amount of time.

□

4.6. The difficulty of replacing an IOR-join

As discussed above, the IOR-join y in Fig. 4.10 cannot be replaced correctly with K-replacement. In this section, we show that the synchronization semantics of the IOR-join in Fig. 4.10 cannot be implemented by a combination of AND and XOR gateways.

Recalling the discussion in Sect. 4.2 we have to specify an equivalence and we require some structure to be preserved in order to rule out some 'simple' implementations that incur an exponential blowup.

In this section, we take the view that the IOR-join synchronizes its incoming branches, where these incoming branches have a certain 'forking' logic, depending on the gateway structure 'before' the IOR-join. The forking logic for the example in Fig. 4.10 is represented by the workflow graph in Fig. 4.11. We will show that the workflow graph in Fig. 4.11 cannot be completed with any combination of AND and XOR gateways to produce a behavior equivalent to the behavior of the workflow graph in Fig. 4.10.

Rather than picking a concrete behavioral equivalence (cf. discussion in Sect. 4.2), we formalize properties that a workflow graph must have to be equivalent to the workflow graph in Fig. 4.10. We allow multiple tasks of the implementing workflow graph to be labeled with z and therefore to correspond to the task z in Fig. 4.10.

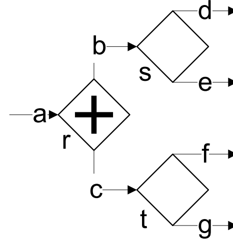


Figure 4.11.: Prefix that cannot be completed.

Definition 4.14 (Equivalent workflow graph properties)**.** *Let W' be a workflow graph which has the prefix illustrated by Fig. 4.11. Let $t_1 = (\{b\}, s, \{d\})$ and $t_2 = (\{c\}, t, \{g\}$. The workflow graph W' satisfies the following properties:*

P1 *There exists no execution where t_1, t_2, and a task labeled with z are executed.*

P2 *There exists an execution during which t_1 and a transition t_z executing a task labeled with z are executed and, for each execution where t_1 and t_z are executed, t_1 is executed before t_z.*

It is easy to see that the workflow graph in Fig. 4.10 satisfies these properties and that notions of equivalence such as the ones that we defined for local and non-local replacements would ensure that any equivalent workflow graph satisfies them, too.

We now present two lemmas that will be useful to prove the workflow graph in Fig. 4.11 cannot be completed:

Lemma 4.15. *Let W be a deadlock-free workflow graph, e, e' be two edges of W, m be a reachable marking of W which marks e.*

If there exists a path p from e to e' in W, then there exists a marking m', reachable from m, such that m' marks e'.

(Can be proved by a straight-forward induction on p. Moreover, Lemma 4.6 implies Lemma 4.15)

Lemma 4.16. *Let W be a deadlock-free acyclic workflow which does not contain IOR-joins. Let $t = (E_1, n, E_2)$ and $t' = (E'_1, n', E'_2)$ be two transitions of W.*

If, in each execution σ of W where t and t' occur, we have that t occurs before t' during σ, then there exists a path from an edge $e \in E_2$ to an edge $e' \in E'_1$.

Proof. We prove this lemma by contradiction: Suppose that in each execution σ of W where t and t' occur we have that t occurs before t' during σ and that there is no path between any edge in E_2 and any edge in E'_1. We show a contradiction by showing the existence of an execution where t' executes before t. We consider the case where n and n' have a single entry and a single exit edge, a similar reasoning can be applied for the other cases.

Let σ be an execution in which t and t' are executed. Let m be the last marking before executing t in σ. The marking m marks the incoming edge e^* of n. Let σ' be the prefix of σ until reaching the marking m. Because t' executes in σ, there exists an edge e, that carries a token in m, and there exists a path p to the outgoing edge e' of n'. By Lemma 4.6, there exist an execution sequence σ'' from m to a marking m'' such that m'' marks e^* and e', i.e, t' was executed during σ'' and t was not. As e^* is marked by m'', the transition t is enabled. Therefore there exists an execution in which t' occurs before t. □

We can now enunciate our result:

Theorem 4.17 (The synchronization role of IOR-joins cannot be implemented using only AND and XOR-logic). *There exists no deadlock-free workflow graph W' such that W' has the workflow graph P illustrated by Fig. 4.11 as prefix, W' does not contain any IOR-join, and W' satisfies the properties of Def. 4.14.*

Proof. We prove this theorem by contradiction: Suppose that there exists a workflow graph W' such that W' has P (illustrated by Fig. 4.11) as prefix, does not contain an IOR-join, and satisfies P1 and P2.

By P2, we have that t_1 and a transition $t_z = (E_1, n, E_2)$ such that n is a task labeled z occur together in some execution and that t_1 always occur before t_z when both occur. By Lemma 4.16, there exists a path p from d to the incoming edge of n.

Consider an execution sequence $\sigma = \langle [a], (\{a\}, r, \{b, c\}), [b, c], t_1, [d, c], t_2, [d, g] \rangle$. We can complete σ to obtain the execution σ^* by following p and thus executing t_z (Lemma 4.15). This contradicts P1 because t_1, t_2, and a task labeled z are executed during σ^*. □

4.7. On the translation to (non-free-choice) Petri nets

We have shown that the synchronization role of an IOR-join cannot always be replaced by a combination of AND and XOR gateways. We have shown in Sect. 2.4 that workflow graphs without IOR gateways correspond to free-choice workflow nets. Hence, there is no free-choice workflow net that can implement the synchronization role of the IOR-join in Fig. 4.10.

To translate an acyclic workflow graph into a non-free-choice Petri net, one can use *dead path elimination* (1, 80) to implement the IOR-join with gateways that have a local semantics. Dead path elimination uses workflow graphs with individual tokens, where a token can have a value that is either *true* or *false*. Such a workflow graph can be easily translated into a high-level Petri net, which in turn can be unfolded into a non-free-choice Petri net.

A more direct approach to implement the IOR-join from Fig. 4.10 as a Petri net is shown in Fig. 4.12, where the IOR-join replacement is delimited by the dashed box. The

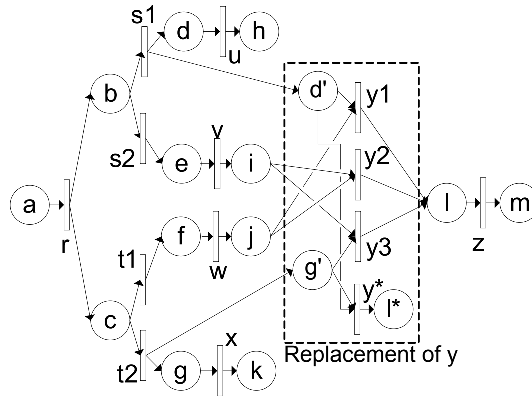


Figure 4.12.: A non-free-choice Petri net that is equivalent to the one in Fig. 4.10.

behavior of this Petri net is equivalent to the workflow graph in Fig. 4.10. Note that, similar to K-replacement, we provide additional inputs to the IOR-replacement and that this construction preserves most of the structure of the original workflow graph. These additional inputs give the IOR-join replacement information on the edges that have been marked by the execution.

We can then think of the IOR-join as two boolean expressions over these marked edges that fully characterize its executions: the first expression characterizes the executions that lead to a token on the outgoing edge of the IOR-join, the second characterizing all other executions. Both expressions can be easily represented by a non-free-choice Petri net as in the example in Fig. 4.12, where the transitions y_1, y_2, and y_3 implement the first expression $(d' \wedge j) \vee (i \wedge j) \vee (i \wedge g')$ The transition y^* implements the second expression. The role of y^* is to "purge" the place d' and g' in the second case.

This construction can be defined to implement an IOR-join in any acyclic workflow graph because the full execution history can be provided to the replacement subgraph. However, the Petri net representations of the boolean expressions are exponential in the size of the workflow graph. We leave it as future work to evaluate whether this exponential blowup can be mitigated using simplification techniques for boolean formulas.

To sum up, Thm. 4.17 gives a strong argument why IOR-joins cannot be easily implemented by a free-choice workflow net, it points to some difficulty when trying to translate to general workflow nets.

4.8. Summary

In this chapter, we studied the difficulty of replacing an IOR logic using AND and XOR logic only. We have shown that, while there exists straightforward translations for IOR-splits, it is more difficult to replace an IOR-join.

For acyclic processes, we established a necessary and sufficient condition that characterizes IOR-joins that can be locally replaced. For the IOR-joins that can be locally replaced, we proposed a generic local replacement. We presented a non-local replacement strategy which allows us to translate some of the IOR-joins that cannot be replaced locally. We have shown that computing this replacement and checking its condition of application can be done in polynomial time with respect to the size of the original workflow graph.

While these results have been presented for the replacement of a single IOR-join in a sound acyclic business process model, they can be applied to replace multiple IOR-joins in an acyclic workflow graph. Moreover, it can be shown that both replacement techniques do not alter the soundness of the process, i.e., cannot introduce or fix a control-flow error which makes these replacement strategies applicable to replace IOR-joins in process models of which the soundness is unknown such as, for example, when performing a control-flow analysis. We will see later (in Sect. 6.2.1) how we can use *process structure trees* to decompose the workflow graph into fragments. Such a decomposition allows us to factor out cycles and therefore to apply the replacements in processes containing cycles[1].

We have shown that the synchronization provided by the IOR-join cannot, in general, be implemented by free-choice constructs. Translations of a workflow graph containing an IOR-join into a (non-free-choice) Petri net exist, however, known translations have an exponential blowup and, usually, do not preserve the structure of the original process. These results show a difficulty to translate the non-local semantics of the IOR-join into a modeling language that only contains local gateways and therefore a difficulty to fully leverage Petri net based techniques for process models containing IOR-joins.

We will discuss in the next chapter a control-flow analysis technique allowing to check the soundness of acyclic workflow graphs containing IOR-joins.

4.9. Related work

To our knowledge, no existing general translation satisfies our three requirements of equivalence, efficiency, and structure preservation simultaneously. But translations for subsets of workflow graphs or translations satisfying a subset of the requirements exist. In the following, we discuss the qualities of these translations and their shortcomings.

Wynn et al. (83) use a full state space exploration to find each IOR-join that has only mutually exclusive incoming edges, and thus can be replaced by an XOR-join. Likewise, an IOR-join that has only pairwise always-concurrent edges is replaced by an AND-join. This requires an exponential time and only works for the rare IOR-joins whose incoming edges meet one of these two conditions. Note that these conditions

[1] Some extra care must be taken when doing because, in some particular cyclic cases, the semantics of workflow graphs containing IOR-joins is not always compositional with respect to the process structure tree fragments (see (80) for more details).

can also be computed in quadratic time for acyclic workflow graphs (26).

Vanhatalo et al. (76) use the *refined process structure tree* to decompose a workflow graph. The decomposition provides local replacement heuristics for some, but not all, IOR-joins. Although this approach is incomplete, the decomposition and heuristics can be computed in linear time and, because they result in local replacements, preserve the structure of the original model.

Mendling et al. (51) use the theory of regions to synthesize a Petri net from the reachability graph of the process model. The resulting net loses the structure of the original process and its size grows exponentially with respect to the size of the original process. While the paper does not prove any notion of equivalence that is preserved, it further points to the work of Cortadella et al. (5), which guarantees that the synthesized net is bisimilar to the process model reachability graph.

As mentioned in the previous section, one could use the dead path elimination mechanism of BPEL (1, 80) to implement IOR-join in acyclic processes. There are two approaches in the literature to translating dead path elimination to high-level Petri nets (47, 63). A high-level Petri-net can be further translated into a low-level Petri net by a well-known unfolding construction, which in general incurs an exponential blowup.

Ouyang et al. (55) also give a translation from BPEL processes to Petri nets. In contrast to the work of Lohmann (47) and Stahl (63), they model dead path elimination directly using low-level nets. The evaluation of an IOR-join is modeled using a Petri-net encoding a boolean function similarly to the idea we presented in Sect. 4.7.

In all three approaches, the resulting Petri net is exponentially larger than the original process model. It can be argued that all three approaches fully preserve the behavior of the original process model and preserve its structure to a large extent. Note that these three translations in general result in a non-free-choice Petri net.

5

Symbolic Execution of Acyclic Workflow Graphs

In this chapter, we propose a new technique to analyze the control-flow, i.e., the workflow graph of a business process model, which we call *symbolic execution*. We consider acyclic workflow graphs that may contain IOR gateways and define a symbolic execution for them that runs in quadratic time. The result allows us to decide in quadratic time, for any pair of control-flow edges or tasks of the workflow graph, whether they are sometimes, never, or always reached concurrently. This has different applications, including finding control-flow errors. In the following, we show how to decide soundness of an acyclic workflow graph with inclusive OR gateways in quadratic time. Moreover, we show that symbolic execution provides diagnostic information that allows the user to efficiently deal with spurious errors that arise due to over-approximation of the data-based decisions in the process. Finally, we use symbolic execution to prove that deciding weak soundness for an acyclic workflow graph containing IOR-joins is NP-hard.

Some control-flow errors can be characterized in terms of relationships between control-flow edges or tasks of the process. For example, a process is free of *deadlock* if any two incoming edges of an AND-join are always marked concurrently. We can say that such a pair of edges is *always concurrent*. A process is free of *lack of synchronization* if any two incoming edges of an XOR-join are *mutually exclusive*, i.e., they never get marked concurrently. A data-flow hazard may arise if two conflicting operations on the same data object are executed concurrently. That can happen only if the tasks containing the data operations are *sometimes concurrent*, i.e., not mutually exclusive. Similar relationships have also been proposed for a behavioral comparison of processes (81).

Such control-flow relations can be computed by enumerating all reachable control-flow states of the process by explicitly executing its *workflow graph*, i.e., its control-flow

representation. However, there can be exponentially many such states, resulting in a worst-case exponential time algorithm. We propose a form of symbolic execution of a workflow graph. We consider acyclic workflow graphs that may contain inclusive IOR gateways and define a symbolic execution of such graphs that runs in quadratic time. It captures enough information to allow us to decide, using a complementing graph analysis technique, the above mentioned relationships for any pair of control-flow edges in quadratic time. In particular, we obtain a control-flow analysis that decides *soundness* in quadratic time for any acyclic graph that may contain IOR gateways.

The symbolic execution keeps track of which decision outcomes within the process flow lead to which edge being marked. Therefore, it can provide information, in case of a detected error, about which decisions potentially lead to the error. We show how this leads to more compact diagnostic information than obtained with prior techniques. In particular, we show how this allows the user to efficiently deal with spurious errors that arise due to over-approximation of the data-based decisions in the process.

A portion of these results were first presented at the 2$^{\text{nd}}$ Central-European Workshop on Services and their Composition (30). The extended results have been presented at the 12th International Conference on Business Process management (BPM 2010) (26). Additionally, a patent (28) on these results is pending.

The rest of this chapter is structured as follows: After introducing the preliminary notions, we introduce symbolic execution in Sect. 5.1 and show how the relationship 'always-concurrent' and the absence of deadlock can be decided. Then, we discuss the 'sometimes-concurrent' and 'mutually-exclusive' relationships and lack of synchronization in Sect. 5.2. In Sect. 5.3, we show how the diagnostic information provided by symbolic execution can be used to deal with the over-approximation that results from abstracting from data-based decisions.

5.1. Symbolic execution and always-concurrent edges

In this section, we introduce *symbolic execution* and show how we use it to detect deadlocks and determine whether two edges are always-concurrent. We start by giving a characterization of deadlock, then introduce the symbols and the propagation rules of the symbols, we show how to compute a normal form of a symbol and discuss the complexity of the proposed technique.

Let, in this section, $W = (N, E, c, l)$ be an acyclic workflow graph prefix that is free of lack of synchronization. We describe in Sect. 5.2.3 how we determine such a prefix.

5.1.1. Equivalence of edges and a characterization of deadlock

A deadlock arises at an AND-join when one of its incoming edges e is marked during an execution σ but another edge e' does not get marked during σ because, as e' never gets marked during σ, the AND-join cannot execute and the token marking e is 'blocked'. Thus, in order to have no deadlock, the incoming edges of an AND-join

need to get marked 'together' in each execution. We can precisely capture this through *edge equivalence* or the notion *always-concurrent*. In an acyclic workflow graph, only an AND-join or an IOR-join can block a token but only an AND-join can '*cause*' a deadlock: An IOR-join can 'block' a token if and only if there exists a preceding node that blocks another token. Thus, whenever there is a deadlock in an acyclic workflow graph, there exists an AND-join with non-equivalent incoming edges.

To introduce edge equivalence, we define the *Parikh vector* (9) of an execution, which records, for each edge, the number of tokens that are produced on that edge during the execution.

Definition 5.1 (Parikh vector)**.** *The* Parikh vector *of an execution* $\sigma = \langle m_0, T_0, ...\rangle$, *written* $\vec{\sigma}$, *is the multi-set of edges such that* $\vec{\sigma}[s] = 1$ *for the source* s *of* W *and otherwise* $\vec{\sigma}[e] = |\{i \mid T_i = (E, n, E') \wedge e \in E'\}|$.

For example, given the execution $\sigma = \langle [s], (\{s\}, F, \{a, b, c\}), [a, b, c], (\{a\}, X, \{d\}), [b, c, d], (\{d\}, Y, \{h\}), [b, c, h], (\{c\}, I, \{f\}), [b, f, h]\ (\{b, f\}, M, \{i\}), [h, i], (\{h, i\}, J, \{j\}), [j]\rangle$ of the workflow graph of Fig. 5.1, we have $\vec{\sigma}[s] = \vec{\sigma}[a] = \vec{\sigma}[b] = \vec{\sigma}[c] = \vec{\sigma}[d] = \vec{\sigma}[f] = \vec{\sigma}[h] = \vec{\sigma}[i] = \vec{\sigma}[j] = 1$ and $\vec{\sigma}[e] = \vec{\sigma}[g] = 0$.

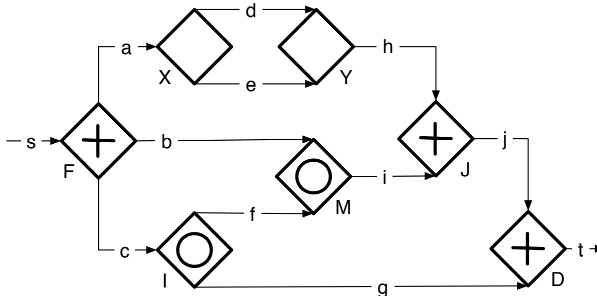


Figure 5.1.: A workflow graph.

Two executions are interleaving equivalent if they execute the same transitions; possibly in a different order, i.e., if they have the same Parikh vector. Edges are sometimes concurrent if they can carry a token simultaneously during an execution. Edges are equivalent if they are marked by the same set of executions, but not necessarily simultaneously:

Definition 5.2 (Edge equivalence, always-concurrent)**.**

- *Two edges are* parallel *in an execution* σ *if there is a marking in* σ *in which both edges are marked.*
- *Two executions* σ, σ' *are* interleaving equivalent *if* $\vec{\sigma} = \vec{\sigma}'$.
- *Two edges are* concurrent *in an execution* σ *if there is an execution* σ' *such that* σ *and* σ' *are interleaving equivalent and the edges are parallel in* σ'.

- *Two edges are* always-concurrent *if they are concurrent in every execution of W.*
- *Two edges e_1 and e_2 of W are* equivalent, *written $e_1 \equiv e_2$, if for any execution σ of W, we have $\vec{\sigma}[e_1] = \vec{\sigma}[e_2]$.*

Note that these definitions are founded only for acyclic workflow graphs.
We now relate edge equivalence and always concurrent:

Proposition 5.3. *Two edges e_1, e_2 are always-concurrent iff $e_1 \equiv e_2$ and $e_1 \parallel e_2$*[1]

Proof. Consider two equivalent edges e_1, e_2 and an execution σ marking e_1 and thus, by equivalence marking e_2. We can assume without loss of generality that e_1 is marked before e_2 in σ. Let us consider the first marking of σ marking e_1. Because there is a future marking in σ where e_2 is marked, there is a path from an edge carrying a token to p_2. The key intuition behind the proof of this proposition is that tokens on unrelated edges can move independently. Indeed we can execute continue to execute the transitions in σ (in a different interleaving) to move the token to e_2 without having to move the token on e_1 because, if we needed to move the token on e_1 to move the other token towards e_2, there would need to be a path from e_1 to e_2 which would contradict the assumption that $e_1 \parallel e_2$. Once the second token reaches e_2, we have shown that there is an interleaving equivalent execution to σ in which e_1 and e_2 are parallels and thus, as this can be done for any execution, that e_1 and e_2 are always concurrent.

Consider two edges e_1, e_2 that are always-concurrent. As e_1 and e_2 are always concurrent $\vec{\sigma}[e_1] > 0$ if and only if $\vec{\sigma}[e_2] > 0$ for any execution. In an acyclic workflow graphs prefix that is free of lack of synchronization, a transition is executed at most once, i.e., for any edge e we have $\vec{\sigma}[e] \leq 1$. Thus, $\vec{\sigma}[e_1] = \vec{\sigma}[e_2] > 0$ in every execution. We now show that $e_1 \parallel e_2$ by contradiction: we can assume, without loss of generality that $e_1 > e_2$. As e_1 and e_2 are always concurrent, there is a reachable marking such that e_1 and e_2 are marked simultaneously. These two marked edges, which are on a simple path to the sink of the workflow graph, can be proven to indicate a lack of synchronization (the proof idea involves moving the token on e_1 until it reaches e_2 as done thoroughly in Lemma 3.11). This lack of synchronization contradicts the hypothesis of an acyclic workflow graphs prefix that is free of lack of synchronization. □

In the workflow graph depicted by Fig. 2.4, we have $a \equiv b \equiv h$ and $a \not\equiv d \not\equiv g$. Note that we have discussed earlier an execution of the workflow graph of Fig. 2.4 where $\vec{\sigma}[a] = \vec{\sigma}[d]$. However, there exists another execution such that $\vec{\sigma}[a] \neq \vec{\sigma}[d]$ and therefore $a \not\equiv d$. Moreover, a is always-concurrent to b but not to h.

Proposition 5.4. *W contains a* deadlock *iff there exist two incoming edges of an AND-join of W that are not equivalent, or equivalently, that are not always-concurrent.*

[1] $e_1 \parallel e_2$ indicates that there is no path from e_1 to e_2 and vice et versa (cf. Def. 2.14)

Proof. The intuition behind the proof of this proposition is that the number of tokens marking two incoming edges of an AND-join must be the same in every sound execution, which is exactly what is captured by the definition of edge equivalence (Def. 5.2). If two incoming edge e_1, e_2 of an AND-join are not equivalent then there exist an execution σ such that, without loss of generality, $\vec{\sigma}[e_1] > \vec{\sigma}[e_2]$. By the workflow graph semantics (Def. 2.18), the AND-join will execute $\vec{\sigma}[e_2]$ leaving $\vec{\sigma}[e_1] - \vec{\sigma}[e_2]$ tokens blocked on e_1 and thus creating a deadlock. If the edges are equivalent, no token stays blocked.

As two incoming edges of an AND-join are always unrelated, for two incoming edges of an AND-join in an acyclic workflow graphs prefix that is free of lack of synchronization the notion of equivalence and always-concurrent are equivalent by Proposition 5.3. ☐

In an acyclic workflow graphs, or an acyclic workflow graph prefix, that is free of lack of synchronization, a transition is executed at most once, i.e., for any edge e we have $\vec{\sigma}[a] \leq 1$. In such a setup, the reciprocal direction of Proposition 5.3 also holds. Therefore, for two incoming edges of an AND-join, which are unrelated by definition, in acyclic workflow graph prefixes that are free of lack of synchronization.

In the workflow graph depicted by Fig. 2.4, the edges j and g are not always-concurrent. Therefore, we get a deadlock at the AND-join D.

In the following, we show how we can compute edge equivalence and therefore also whether two edges are always-concurrent.

5.1.2. Symbolic execution

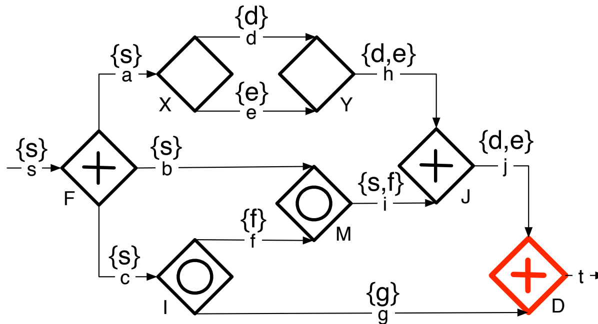


Figure 5.2.: The assignment resulting from the symbolic execution of the workflow graph of Fig. 2.4.

The first step to compute edge equivalence is the symbolic execution of the workflow graph. During symbolic execution, each edge is labeled with a symbol, which is a set of *outcomes* of the workflow graph. An *outcome* is the source edge of the workflow graph, an outgoing edge of an XOR-split, or an outgoing edge of an IOR-split in the graph. Figure 5.2 shows the labeling of the workflow graph of Fig. 2.4 that results from its symbolic execution.

The intuition behind symbolic execution is to label an edge e with a set S of outcomes such that e is marked during an execution σ if and only if some of the outcomes in S get marked during σ.

The symbolic execution starts with labeling the source s with $\{s\}$. All other edges are yet unlabeled. If all incoming edges of a node are labeled, we may label the outgoing edges of the node by applying one of the propagation rules depicted by Fig. 5.3, depending on the logic of the node.

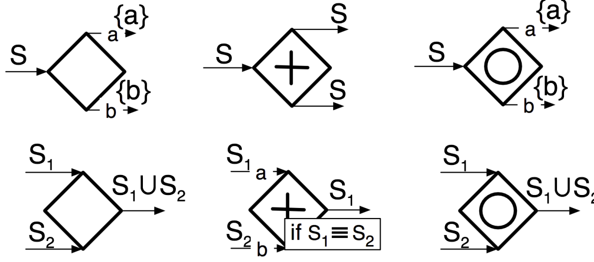


Figure 5.3.: The propagation rules.

In general, the label of the outgoing edges depends on the labels of the incoming edges. However, if the node is an XOR-split or an IOR-split, then the symbol that is assigned to one of the outgoing edges only contains that outgoing edge. The symbol associated to the incoming edge of the node is then ignored. In case of an AND-join, the propagation rule additionally requires the symbol labeling its incoming edges to be equivalent (which we will describe in Sect. 5.1.3) in order to be applied. The AND-join rule then chooses one of the labels of the incoming edges non-deterministically as the label for the outgoing edge (Two equivalent symbols are not necessarily equal like for example the symbols on the edges h and j of Fig. 5.3) . The symbol labeling an outgoing edge of a node that is an XOR-join or an IOR-join, is the union of the symbols labeling the incoming edges of the node. The symbolic execution terminates when there is no progation rule that can be applied. In the following, we define these propagation rules formally.

Definition 5.5 (Symbolic execution)**.** *An* outcome *of W is the source, an outgoing edge of some XOR-split, or an outgoing edge of some IOR-split of W. A* symbol *of W is a set of outcomes of W. An* assignment *is a mapping φ that assigns a symbol to each edge of some prefix of W. If e is an edge of that prefix, we say that e is* labeled under *φ.*

For every node n of a workflow graph, we describe the propagation *by the node n from an assignment φ to an assignment φ', written $\varphi \xrightarrow{n} \varphi'$. The propagation $\varphi \xrightarrow{n} \varphi'$ is* activated *when all the incoming edges of n are labeled under φ and no outgoing edge is labeled under φ. Additionally, if n is an AND-join, the symbol associated to each incoming edges of n must be equivalent (according to Def. 5.7) for the propagation to*

be activated. If n is activated in φ, we have $\varphi \xrightarrow{n} \varphi'$ where φ' is obtained as follows, for any edge e of W:

- *If $l(n) = AND$ and there exists an edge $e' \in \circ n$, then $\varphi'(e) = \varphi(e')$ for $e \in n\circ$ and $\varphi'(e) = \varphi(e)$ otherwise.*
- *If n is an XOR-split or an IOR-split, then $\varphi'(e) = \{e\}$ for $e \in n\circ$ and $\varphi'(e) = \varphi(e)$ otherwise.*
- *If n is an XOR-join or an IOR-join, for $\varphi'(e) = \bigcup_{e' \in \circ n} \varphi(e')$for $e \in n\circ$ and $\varphi'(e) = \varphi(e)$ otherwise.*

As said above, the propagation rules establish that an edge e is marked during an execution σ if and only if some of the outcomes in $\varphi(e)$ are marked during σ:

Lemma 5.6. *For any execution σ of W and any edge $e \in E$, $\vec{\sigma} \otimes \varphi(e) > 0 \Leftrightarrow \vec{\sigma}[e] > 0$.*

Proof. We perform an induction on the number of edges labeled under φ. We increase the number of edges labeled under φ using propagation rules of Def. 5.5.

Base case: Only the source edge s of W is labeled under φ with the symbol $\{s\}$. By the definition of the Parikh vector (Def. 5.1), $\vec{\sigma}[s] = 1$. Trivially, we have $\vec{\sigma} \otimes \varphi(s) = 0 \Leftrightarrow \vec{\sigma}[s] = 0$.

Induction step: Assuming an assignment φ of W according to Def. 5.5 and that $\varphi(e) \otimes \vec{\sigma} = 0 \Leftrightarrow \vec{\sigma}[e] = 0$ holds for each execution σ of W and edge e labeled under φ. We want to show that, when labeling the outgoing edge(s) of any node $n \in N$ according to the assignment propagation rules, i.e., applying one transition rule to increase φ, $\vec{\sigma} \otimes \varphi(e) = 0 \Leftrightarrow \vec{\sigma}[e] = 0$ for each execution σ of W and labeled edge e. As the assigned symbols do not change, we only have to show the latter proposition for the freshly labeled edges.

For each execution σ, each incoming edge $e_I \in \circ n$, and each outgoing edge $e_O \in n\circ$ of n, we consider the four following cases:

1. When n is an XOR-split or IOR-split: By the assignment propagation rules (Def. 5.5), we have $\varphi(e_O) = [e_O]$. Thus, it follows directly from the definition of the Parikh vector (Def. 5.1) that $\varphi(e_O) \otimes \vec{\sigma} = \vec{\sigma}[e_O]$.
2. When n is an XOR-join:

   $\varphi(e_O) \otimes \vec{\sigma} = 0$

   $\Leftrightarrow \sum_{e_{Ik} \in \circ n} \varphi(e_{Ik}) \otimes \vec{\sigma} = 0$, by the assignment propagation rules (Def. 5.5).

   $\Leftrightarrow \sum_{e_{Ik} \in \circ n} \vec{\sigma}[e_{Ik}] = 0$, by the induction hypothesis.

   $\Leftrightarrow \vec{\sigma}[e_O]$, by the workflow graph semantics.

3. When n is an IOR-join:

   $\Leftrightarrow \varphi(e_O) \otimes \vec{\sigma} = 0$,

   $\Leftrightarrow \bigcup_{e_I \in \circ n} (\varphi(e_I)) \otimes \vec{\sigma} = 0$, by the deadlock assignment propagation rule for the IOR-join (Def. 5.5).

   $\Leftrightarrow$ for each edge $e_I \in \circ n$, $\varphi(e_I) \otimes \vec{\sigma} = 0$

   $\Leftrightarrow \sum_{e_I \in \circ n} (\varphi(e_I) \otimes \vec{\sigma}) = 0$.

   $\Leftrightarrow \sum_{e_I \in \circ n} (\vec{\sigma}[e_I]) = 0$, by the induction hypothesis.

   $\Leftrightarrow \vec{\sigma}[e_O] = 0$, by the workflow graph execution semantics and the assignment propagation rules, which requires that there is no deadlock in the labeled prefix of W because, for each AND-join j in the prefix, all the incoming edges of j are equivalent.

4. When $l(n) = AND$:

   $\varphi(e_O) \otimes \vec{\sigma} = 0$

   $\Leftrightarrow \varphi(e_I) \otimes \vec{\sigma} = 0$ because, by the assignment propagation rules (Def. 5.5), there exist an edge e_I such that $\varphi(e_O) = \varphi(e_I)$.

   $\Leftrightarrow \vec{\sigma}[e_I] = 0$, by the induction hypothesis.

   $\Leftrightarrow \vec{\sigma}[e_O] = 0$, because the workflow graph semantics implies that $\Leftrightarrow \vec{\sigma}[e_0] = \min_{e_I \in \circ n} (\vec{\sigma}[e_I])$. Moreover, the assignment propagation rules requires all edges $e \in \circ n$ to be equivalent so that the propagation is activated. Thus, we have $\vec{\sigma}[e_I] = \vec{\sigma}[e_O] = 0$

□

5.1.3. A normal form for symbols

To detect a deadlock or to label the outgoing edge of an AND-join, we need to check edge equivalence. If two incoming edges of an AND-join are not equivalent, we have found a deadlock.

We will exploit that the equivalence of edges corresponds to an equivalence of the symbols they are labeled with. This symbol equivalence can be defined as follows:

Definition 5.7 (Symbol equivalence)**.** *Two symbols S_1, S_2 are* equivalent *w.r.t. W, written $S_1 \equiv S_2$ if, for any execution σ of W, $S_1 \otimes \vec{\sigma} = 0 \Leftrightarrow S_2 \otimes \vec{\sigma} = 0$.*

As W is free of lack of synchronization, for any edge e and for any execution σ, we have $\vec{\sigma}[e] = 1$ or $\vec{\sigma}[e] = 0$. Thus, given two edges e_1, e_2 labeled under φ, the edges e_1 and e_2 are equivalent if and only if the symbols $\varphi(e_1)$ and $\varphi(e_2)$ are equivalent.

We will decide the equivalence of two symbols by computing a normal form for each of them. The normal form of a symbol S is the 'largest' set of outcomes that is equivalent to A. Two symbols are equivalent if and only if they have the same normal form. To show this, we define:

Definition 5.8 (Maximal equivalent extension, Closure)**.** *Let φ be an assignment of W and e be an edge such that e is labeled under φ. Let O be the set of outcomes of W that are labeled under φ.*

- *A* maximal equivalent extension *of $\varphi(e)$ w.r.t. φ is a set $\varphi^*(e) \subseteq O$ such that $\varphi^*(e) \equiv \varphi(e)$ and there exist no other set $S \subseteq O$ such that $\varphi^*(e) \subsetneq S$ and $S \equiv \varphi(e)$.*
- *The* closure *of $\varphi(e)$ w.r.t. φ is the smallest set $\overline{\varphi}(e)$ such that $\varphi(e) \subseteq \overline{\varphi}(e)$ and for each XOR- or IOR-split n such that e' is labeled under φ for each $e' \in n\circ$, we have $\varphi(^\circ n) \subseteq \overline{\varphi}(e)$ iff $n\circ \subseteq \overline{\varphi}(e)$.*

The existence of a maximal equivalent extension is clear. We can also show that it is unique.

Lemma 5.9. *Let φ be an assignment of W and e an edge that is labeled under φ. Then $\varphi^*(e)$ is unique.*

Proof. There exists some maximal equivalent extension of $\varphi(e)$ because e is labeled under φ and $\varphi(e)$ is trivially equivalent to $\varphi(e)$. Thus some outcomes can be added to $\varphi(e)$ until obtaining a maximal equivalent extension. As the number of outcomes is finite, the size of the maximal equivalent extension is finite.

We show that $\varphi^*(e)$, the maximal equivalent extension of $\varphi(e)$, is unique by contradiction: Suppose that there exist two sets of outcomes S_1 and S_2 such that $S_1 \neq S_2$ and S_1 and S_2 are both maximal equivalent extensions of $\varphi(e)$.

Because $S_1 \neq S_2$, we can assume without loss of generality that there exists an edge $e' \in S_2 \setminus S_1$. We show that $S_1 \cup S_2 \equiv \varphi(e)$ which contradicts the maximality of S_1.

As S_1 and S_2 are maximal equivalent extensions of $\varphi(e)$ (Def. 5.8), $S_1 \equiv \varphi(e)$ and $S_2 \equiv \varphi(e)$. It follows from the definition of equivalence (Def. 5.7) that, for any execution σ of W, $S_1 \otimes \vec{\sigma} = 0 \Leftrightarrow \varphi(e) \otimes \vec{\sigma} = 0$ and $S_2 \otimes \vec{\sigma} = 0 \Leftrightarrow \varphi(e) \otimes \vec{\sigma} = 0$. Thus, by definition of the scalar product, for any execution σ of W, $\varphi(e) \otimes \vec{\sigma} = 0 \Leftrightarrow (\sum_{e_1 \in S_1} \{e_1\} \otimes \vec{\sigma} = 0) \wedge (\sum_{e_2 \in S_2} \{e_2\} \otimes \vec{\sigma} = 0)$. Moreover, by definition of the scalar product $(S_1 \cup S_2) \otimes \vec{\sigma} = 0 \Leftrightarrow \sum_{e^* \in S_1 \cup S_2} \{e^*\} \otimes \vec{\sigma} = 0$. Thus, for any execution σ of W, $\varphi(e) \otimes \vec{\sigma} = 0 \Leftrightarrow (S_1 \cup S_2) \otimes \vec{\sigma} = 0$ which is the definition of $\varphi(e) \equiv S_1 \cup S_2$ (Def. 5.7). ☐

It is clear that the closure exists and is unique. Moreover, we can prove that the closure is equal to the maximal equivalent extension:

Lemma 5.10. *Let $W = (N, E, c, l)$ be an acyclic workflow graph, φ be an assignment of W, and $e \in E$ be an edge labeled under φ. We have $\overline{\varphi}(e) \equiv \varphi(e)$*

Proof. By the workflow graph execution semantics we have, $\vec{\sigma}[{}^\circ d] = 0 \Leftrightarrow \sum_{e \in d\circ}(\vec{\sigma}[e]) = 0$. Thus, by Lemma 5.6, $\varphi({}^\circ d) \otimes \vec{\sigma} = 0 \Leftrightarrow \bigcup_{e \in d\circ} \varphi(e) \otimes \vec{\sigma} = 0$. Which, by the assignment propagation rules (Def. 5.5), is equivalent to $\varphi({}^\circ d) \otimes \vec{\sigma} = 0 \Leftrightarrow (d\circ) \otimes \vec{\sigma} = 0$. To compute the closure, we add $d\circ$ if the symbol contains $\varphi({}^\circ d)$ or we add $\varphi({}^\circ d)$ if the symbol contains $d\circ$, it is clear that we stay in the same equivalence class at each operation needed to obtain the closure. □

We can now prove that the closure is equal to the maximal equivalent extension:

Theorem 5.11. *Let φ be an assignment of W. For every edge e that is labeled under φ, we have $\varphi^*(e) = \overline{\varphi}(e)$.*

Proof. For the proof of Thm. 5.11, we need to formulate the following definition and lemma:

We prove that $\varphi^*(e) = \overline{\varphi}(e)$. By Lemma 5.10, $\overline{\varphi}(e) \equiv \varphi(e)$. Thus any edge in $\overline{\varphi}(e)$ is also in $\varphi^*(e)$ by definition of the maximal equivalent extension of $\varphi(e)$ (Def. 5.8). It is left to show that each edge in $\varphi^*(e)$ is also in $\overline{\varphi}(e)$.

We prove by contradiction that there exists no edge e' such that $e' \in \varphi^*(e)$ and $e' \notin \overline{\varphi}(e)$, i.e., $\varphi^*(e) \setminus \overline{\varphi}(e) \neq \emptyset$.

Suppose that there exists an edge e' such that $e' \in \varphi^*(e) \setminus \overline{\varphi}(e)$.

We show how to build an execution σ of W such that $\vec{\sigma} \otimes \varphi^*(e) > 0$ and $\vec{\sigma} \otimes \overline{\varphi}(e) = 0$. The existence of such an execution shows that $\varphi^*(e) \not\equiv \overline{\varphi}(e)$ (Def. 5.7), which contradicts $\varphi^*(e) \equiv \varphi(e)$ because $\overline{\varphi}(e) \equiv \varphi(e)$ by Lemma 5.10.

The construction of σ is performed in two steps: First, we define a path p from the source to e'. Second, we construct the execution σ itself. The construction strategy is to avoid taking any outcome in $\overline{\varphi}(e)$ and thus $\vec{\sigma} \otimes \overline{\varphi}(e) = 0$. The execution follows p to ensure that a token reaches the edge e' and thus $\vec{\sigma} \otimes \varphi^*(e) > 0$.

1. The path p is defined inductively from its last edge e. The induction stops when reaching s. For each edge e_p of p, we ensure that $\varphi(e_p) \setminus \overline{\varphi}(e) \neq \emptyset$.

   As $e' \in \varphi^*(e)$, e' is an outcome, i.e., e' is the outgoing edge of an IOR-split or an XOR-split. By the assignment propagation rules (Def. 5.5), $\varphi(e') = \{e'\}$. Therefore $\varphi(e') \setminus \overline{\varphi}(e) \neq \emptyset$ by assumption.

   The previous elements on the path is defined based on the current element as follows:

   - if the current element is an edge e_c, the previous element is the source node of e_c.

- if the current element is a node n, the previous element is any of the incoming edge e_i of n such that $\varphi(e_i) \setminus \overline{\varphi}(e) \neq \emptyset$.

  We show that an incoming edge e_i of n such that $\varphi(e_i) \setminus \overline{\varphi}(e) \neq \emptyset$ always exists. By induction hypothesis of p there exists an outgoing edge e_o of n such that $\varphi(e_o) \setminus \overline{\varphi}(e) \neq \emptyset$. Let's consider three cases based on the type of n:

  a) If $l(n) = AND$: By the assignment propagation rules (Def. 5.5), there exists an incoming edge e_i of n such that $\varphi(e_i) = \varphi(e_o)$.

  b) XOR and IOR joins: By the assignment propagation rules (Def. 5.5), $\varphi(e_o) = \bigcup_{e_i \in \circ n} \varphi(e_i)$. Thus there exists an incoming edge e_i of n such that $\varphi(e_i) \setminus \overline{\varphi}(e) \neq \emptyset$.

  c) XOR and IOR splits: The incoming edge e_i is such that $\varphi(e_i) \setminus \overline{\varphi}(e) \neq \emptyset$. If it was not the case, then $\varphi(e_i) \subseteq \overline{\varphi}(e)$ and, by the definition of the closure (Def. 5.8), $\varphi(e_o) \subseteq \overline{\varphi}(e)$ because if $\varphi(e_i) \subseteq \overline{\varphi}(e)$, then $n\circ \in \overline{\varphi}(e)$, which contradicts $\varphi(e_o) \setminus \overline{\varphi}(e) \neq \emptyset$

By the definition of workflow graph, for every edge e_g of W, there exists a path from s to e_g. As W is acyclic, we can always build the finite path p starting from s to e' such that, for every edge e_p of p, $\varphi(e_p) \setminus \overline{\varphi}(e) \neq \emptyset$.

2. We build inductively an execution σ such that $\vec{\sigma} \otimes \overline{\varphi}(e) = 0$ and $\vec{\sigma} \otimes \varphi^*(e) > 0$. The insight in building σ is to not mark any edge in $\overline{\varphi}(e)$ and to mark e'. For each marking m of σ, we maintain that every edge e_σ marked in m $\varphi(e_\sigma) \setminus \overline{\varphi}(e) \neq \emptyset$.

   The execution starts with the initial marking $[s]$. As $\varphi(s) = \{s\}$ and s is on p, $s \notin \overline{\varphi}(e)$ and thus $\varphi(s) \setminus \overline{\varphi}(e) \neq \emptyset$.

   We define how any activated node n is executed during σ. The marking m changes to the marking m' as follows:

   - If n has AND logic, is an XOR-join, or an IOR-join: There is a unique execution step possible and the marking changes from m to m' according to the workflow graph semantics. Executing such type of node cannot mark an edge in $\overline{\varphi}(e)$ because, by the assignment propagation rules (Def. 5.5), edges in $\overline{\varphi}(e)$ are exclusively outcomes.
   - If n is an XOR-split or an IOR-split: We distinguish two cases:

     a) If there is one of the outgoing edges e_o of n that is on p, then $m' = m - [{}^\circ n] + [e_0]$, i.e., executing n propagates the token from ${}^\circ n$ to e_0. By construction of p, $e_o \notin \overline{\varphi}(e)$.

     b) If there is no outgoing edge of n that is on p, we pick any outcome e_o such that $e_o \notin \overline{\varphi}(e)$. Again $m' = m - [{}^\circ n] + [e_0]$. By induction hypothesis

$\varphi(^\circ n) \not\subseteq \overline{\varphi}(e)$. Therefore, there is always an outgoing edge e_o of n such that $e_o \notin \overline{\varphi}(e)$ because, by definition of the closure (Def. 5.8), if for any edge $n\circ \subseteq \overline{\varphi}(e)$, then $\varphi(^\circ n) \subseteq \overline{\varphi}(e)$, which contradicts $\varphi(^\circ n) \setminus \overline{\varphi}(e) \neq \emptyset$.

□

That is, we obtain a unique normal form that is equivalent with a given label of an edge. We show in Sect. 5.1.4 that the closure can be computed in linear time. Thus, from the characterization as a closure, we can compute the normal form in linear time. Moreover, the normal form has the desired property:

Theorem 5.12. *$\overline{\varphi}(e) = \overline{\varphi}(e')$ whenever e and e' are equivalent.*

We are now able to compute the closure for the edges g, h, i, and j of the example from Fig. 5.2. We have $\overline{\varphi}(g) = \{g\}$, $\overline{\varphi}(h) = \overline{\varphi}(i) = \overline{\varphi}(j) = \{s, d, e, f, g\}$. As $\overline{\varphi}(h) = \overline{\varphi}(i)$, h and i are equivalent. Thus, there is no deadlock at the AND-join J. On the contrary, $\overline{\varphi}(g)$ differs from $\overline{\varphi}(j)$ which implies, by Thm. 5.12, that g and j are not equivalent. Therefore, we detect a deadlock located at the AND-join D.

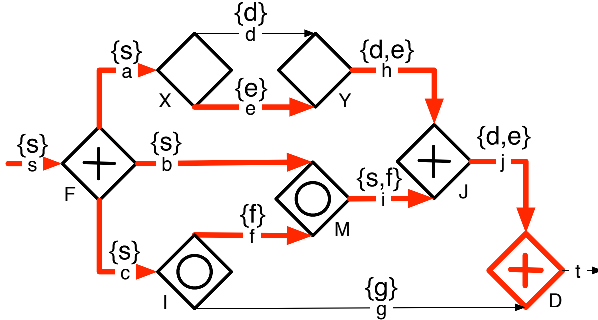


Figure 5.4.: Display of a deadlock.

When we detect a deadlock because two incoming edges of an AND-join are not equivalent, we say that the AND-join is the *location* of the deadlock. To display the deadlock, we can, based on the assignment, generate in linear time an execution, called *error trace*, that exhibits the deadlock. Figure 5.4 depicts how we would display a deadlock: we highlight the location of the deadlock and the error trace, i.e., the edges marked during the execution leading to the deadlock. We discuss in Sect. 5.3 a form of diagnostic information and user interaction that goes beyond this error trace.

5.1.4. Complexity of the computation

In this section, we first describe an approach to compute the closure in linear time and then discuss overall the complexity of symbolic execution.

Let, in this section, φ be an assignment of W and D be the set of IOR-splits and XOR-splits of W such that, for every node $d \in D$, every edge in $d\circ$ is labeled under φ. We define a *closure operation of* a node $d \in D$ *on* a symbol S that changes S *to* a symbol S' such that $S' = S \cup \varphi(^\circ d) \cup d\circ$. A closure operation of a node n changing S to S' is enabled when $\varphi(^\circ d) \in S$ or $d\circ \in S$ and $S \neq S'$. The computation of the closure comprises two phases:

1. We go through the nodes from the maximal to the minimal element in D w.r.t. the precedence relation $<$, i.e., from the right most nodes in the graph to the left most nodes of the graph. For each node n, we execute the closure operation of n if it is enabled.

2. We go through the nodes from the minimal to the maximal element in D w.r.t. the precedence relation $<$. For each node n, we execute the closure operation of n if it is enabled.

It is clear that this computation requires linear time. Note that D must be sorted. This sorting is obtained once and for all before performing symbolic execution and requires linear time with respect to the size of the workflow graph. Moreover, we show that this sequence of phases is sufficient to ensure completeness of the closure computation:

Lemma 5.13. *After performing phase 1 and phase 2 on a symbol S, there exists no node $n \in D$ such that a closure operation of d is enabled on S.*

Proof. We have to ensure that after the two phases there is no more closure operation that is enabled:

We separate the enablement condition of a closure operation in two part and say that a closure operation of a node $d \in D$ on a symbol S to S' is *right enabled* iff $d\circ \subseteq S$ and $S \neq S'$. It is *left enabled* iff $\varphi(\circ(d)) \subseteq S$ and $S \neq S'$.

A closure operation of a node d is enabled only once because, by definition of left enabled and right enabled, an operation is enabled only if $S \neq S'$, a closure operation grows S by adding to it $\varphi(\circ d) \cup d\circ$, and S is monotonously growing during the computation of the closure.

We establish four propositions that help us in the proof:

1. A right enabled closure operation of $d \in D$ on S to S' cannot right enable a closure operation of a node $d' \in D$ such that $d < d'$ because a right enabled closure operation of d only adds to the symbol outcomes that precede d.

2. Similarly, a left enabled closure operation of $d \in D$ on S to S' only adds outcomes that follow d. Thus, it cannot left enable a closure operation of a node that precedes d.

3. We now show that a left enabled closure operation cannot right enable a closure operation: Assume that we perform a left enabled closure operation of a noded $d \in D$ on S to S'. We have $S' - S \cup d\circ$. For any $d' \in D$ such that $d \neq d'$, we have $d \circ \cap d' \circ = \emptyset$. Thus, the operation cannot right enable a closure operation of d'. Moreover, it cannot right enable a closure operation of d because a closure operation of a node cannot be nabled twice.

4. By 1 and 3, we have that a closure operation of a node d can only right enable a node that precedes d.

We now show by contradiction that at the end of phase 1 all the right enabled closure operations are performed, i.e, no closure operation is right enabled and it is not possible to right enable a closure operation: Assume that there exists a right enabled closure operation at the end of phase 1. Consider the node d that is the minimal node with respect to $<$ (,i.e., the left most node) such that a closure operation of d is right enabled at the end of phase 1. As a closure operation of d is enabled only once, the closure operation was not enabled when phase 1 visited d. Thus, a closure operation on a node preceding d enabled right the closure operation of d, which is in contradiction with 4. We have shown that there is no closure operation that is right enabled at the end of phase 1. Moreover, by 3 we have that there will not be right enabled closure operation anymore.

We now show that at the end of phase 2 there is no more left enabled closure operation. Again, we proceed by contradiction: Assume that at the end of phase 2 there exists a left enabled closure operation. Consider the node d that is the maximal node with respect to $<$ (,i.e., the right most node) such that a closure operation of d is left enabled at the end of phase 2. As a closure operation of d is enabled only once, the closure operation was not enabled when phase 2 visited d. Thus, a closure operation on a node following d enabled left the closure operation of d, which is in contradiction with 2 and that there will not be right enabled closure operation anymore. □

Symbolic execution needs just one traversal of the workflow graph. The closure is the most expensive operation. We have shown how to compute the closure of a symbol in linear time with respect to the size of D. Therefore, each transition takes at most linear time and the overall worst-case time complexity is quadratic.

5.2. Lack of synchronization and sometimes-concurrent edges

The workflow graph depicted in Fig. 5.5 permits the execution $\sigma = \langle [s], [a, b], [b, d], [d, e], [e, g], [g, g], ... \rangle$. The edge g is marked by two tokens in the marking $[g, g]$. Thus, the workflow graph depicted by Fig. 5.5 contains a lack of synchronization. In this section, we describe an algorithm that detects lack of synchronization and sometimes-concurrent edges in acyclic workflow graphs. The technique has quadratic time complexity.

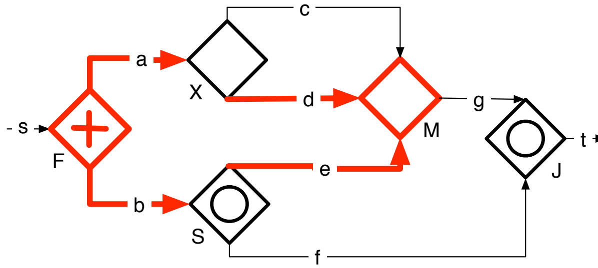


Figure 5.5.: A workflow graph that contains a lack of synchronization.

We first give a characterization of lack of synchronization in terms of *acyclic T/P handles* that we will simply call *handles* for simplicity. Then we show how these handles can be computed in quadratic time. We describe how to combine the symbolic execution and handle detection to detect control-flow errors. Finally, we show how to compute whether two edges are sometimes-concurrent, which has separate applications such as data-flow analysis.

5.2.1. Handles and lack of synchronization

In Chap. 3, we gave a structural characterization of soundness for free-choice workflow nets. This characterization prohibits a simple path to the final place to have a T/P handle in a sound workflow net. For acyclic workflow graphs, which may contain IOR-join, we show that a similar pattern characterizes lack of synchronization.

We follow the intuition that paths starting with an IOR-split or an AND-split, should not be joined by an XOR-join. In the following, we formalize this characterization and show that these structures are the sources, and the only sources, of a lack of synchronization in deadlock-free acyclic workflow graphs.

Definition 5.14 (Path with an AND/XOR or an IOR/XOR handle)**.** *Let* $p_1 = \langle n_0, ..., n_i \rangle$ *and* $p_2 = \langle n'_0, ..., n'_j \rangle$ *be two paths in a workflow graph* $W = (N, E, c, l)$*.*

The paths p_1 *and* p_2 *form a* path with a handle[1] *if* p_1 *is not trivial,* $p_1 \cap p_2 = \{n_0, n_i\}$*,* $n_0 = n'_0$*, and* $n_i = n'_j$*. We say that* p_1 *and* p_2 *form a path with a handle* from n_0 to n_i*. We speak of a* path with an IOR-XOR handle *if* n_0 *is an IOR-split and* n_i *is an XOR-join. We speak of a* path with an AND-XOR handle *if* n_0 *is an AND-split, and* n_i *is an XOR-join. In the rest of this document, we use handle instead of path with an AND-XOR handle or path with an IOR-XOR handle. The node* n_0 *is the* start node *of the handle and the node* n_i *is the* end node *of the handle.*

Definition 5.15 (Minimal handle)**.** *A handle* h *from* n_0 *to* n_i *is* minimal *iff there exists no other handle* h' *from* n'_0 *to* n'_j *such that* $n'_j < n_i$*, or* $n_i = n'_j$ *and* $n_0 < n'_0$*.*

[1]Strictly speaking, one path is the handle of the other path and vice versa.

The relationship between our notion of handle in a workflow graph and a path to the final place with a T/P handle in a workflow net is the following: consider any acyclic workflow graph without IOR-joins that contains an AND/XOR handle. For simplicity, assume that this workflow graph has a single sink. Let N be the workflow net obtained by translating the workflow graph using the translation presented in Sect. 2.4. Let p_1 and p_2 the two paths corresponding to the handle in N. The paths p_1 and p_2 will both start at a transition t and end in a place p and be disjoint elsewhere by construction. As N is acyclic, there exists a path p_3 from p to the final place of N that is disjoint from p_1 and p_2. Thus p_1 is a T/P handle of the simple path to the final place obtained by the concatenation of p_1 and p_2.

We now prove that a handle characterizes lack of synchronization:

Theorem 5.16. *In an acyclic workflow graph that contains no deadlock, there is a lack of synchronization iff there is a handle.*

Proof. Let an acyclic workflow graph $W = (N, E, c, l)$ that does not contain any deadlock.

$\Rightarrow$

We show that if there is a lack of synchronization in W, then there is a handle in W.

Assume that there is a lack of synchronization in W, i.e., there is a reachable marking m^* such that an edge is marked with more than one token in m^*.

Therefore, there exists a reachable marking m such that there is an edge $e \in E$ that a is marked with more than one token in m and there exists no other marking m' such that an edge $e' \in E$ precedes e and is marked with more than one token in m'.

By the definition of reachable marking, there exists an execution σ of W such that $m \in \sigma$.

By the semantics of workflow graphs, the source node n of e is an XOR-join. n cannot be another type of node as if it was of another type n would have been executed twice and thus a reachable marking m' would exist with an edge e' preceding e such that e' is marked with more than one token in m'.

By the semantics of workflow graphs and as none of the edges preceding e can be marked with more than one token, there has to be two edges $e_1, e_2 \in \circ n$ that are marked with one token in σ.

We use σ to define two paths p_1 and p_2 in W such that every edge in p_1 or p_2 is marked during σ. The path p_1 and p_2 start at the source edge of W. The path p_1 ends with $\langle e_1, n \rangle$, p_2 with $\langle e_2, n \rangle$. We define p_1 recursively from e_1, p_2 recursively from e_2.

Each node $n_k \in p_1$ is defined by the edge e_k that follows n in p_1 such that n_i is the source of e_i. Each edge $e_{k-1} \in p_1$ is defined by the node n_k that follows e_{k-1} in p_1 as

follows:

$$e_{k-1} = \begin{cases} \circ n_k & \text{if } n_k \text{ is an XOR-split, IOR-split, or an AND-split, (1)} \\ e_n & \text{if } n_k \text{ is an XOR-join or an IOR-join, } e_n \in \circ n_k\text{, and there exists a} \\ & \text{marking } m_n \text{ in } \sigma \text{ such that } e_n \text{ is marked in } m_n\text{, (2)} \\ e_n & \text{if } n_k \text{ is an AND-join and } e_n \in \circ n_k\text{. (3)} \end{cases}$$

In case (1), p_1 contains the only incoming edge of n_k. It follows from the workflow graph semantics that if an outgoing edge of n_k is marked during σ, then the incoming edge of n_k is marked during σ.

In case (2), p_1 contains e_n that was marked during σ. By the workflow graph semantics, e_n exists because there is a token on the outgoing edge of n_k.

In case (3), p_1 contains any of the incoming edges of n_k. By the workflow graph semantics, every incoming edge of n_k is marked during σ because the outgoing edge of the n_k is marked during σ.

Each node and edge of p_2 is defined similarly to p_1 using e_2 instead of e_1 as basis for the recursion.

There exists a node f such that f belongs to p_1 and p_2, and there exists no node, other than n, that follows f and that belongs to p_1 and p_2.

It follows from the workflow graph semantics that f is an AND-split or an IOR-split because two of its outgoing edges get marked during σ and, by assumption, there is no edge preceding e that is marked with more than one token in any execution of W.

Thus, there exist two paths between the AND-split f and the XOR-join n that are disjoint aside from f and n. By Def. 5.14 they form a handle.

$\Leftarrow$

We show that if there is a handle in W, then there is a lack of synchronization in W.

Assume that there is a handle in W, i.e., there exist two paths $p = \langle n_0, e_0, ..., e_{n-1}, n_n \rangle$ and $p' = \langle n'_0, e'_0, ..., e'_{n-1}, n'_k \rangle$ such that n_0 is an AND-split or an IOR-split, n_n is an XOR-join, $n_0 = n'_0$, $n_n = n'_k$ and the two paths are disjoint aside from n_0 and n_n (Def. 5.14).

By the definition of workflow graphs, there is a path from the source edge to n_0. Thus, as there is no deadlock, a marking m_0 where the incoming edge of n_0 is marked is reachable in W.

By the semantics of workflow graph, n_0 can execute and its execution can result in a marking m_1 where e_0 and e'_0 are marked.

There exists a set of marking M reachable from m_1 such that for each marking $m \in M$ there exist an edge of p and an edge of p' that are marked in m.

We show that there exists a reachable marking in M such that e_{n-1} and e'_{n-1} are both marked.

We show for any m_i in M such that $m_i[e_{n-1}] = 0$ or $m_i[e'_{n-1}] = 0$, there exists a transition that changes m_i into a marking m_{i+1} that belongs to M.

As there is no deadlock, there is always at least one node n that is activated in m_i, such that $n \neq n_n$. Note that if n_n is activated in m_i, there exists another node n activated in m_i. This is because, by definition of m_i, there is an edge e_j marked prior to e_{n-1} or e'_{n-1} and, as there is no cycle, there is no path from n_n to the target t of e_j. Thus, the execution of t cannot require the prior execution of n_n and, as there is no deadlock, either t is activated in m_i or a node prior to t is activated in m_i.

By assumption on m_i, there exists an edge e_i on p and an edge e'_i on p' such that both are marked in m_i and either $e_i \neq e_{n-1}$ or $e'_i \neq e'_{n-1}$.

We distinguish three cases:

1. $e_i \notin \circ n$ and $e'_i \notin \circ n$

   Executing n in m_i results in a marking m_{i+1}. By the workflow graph semantics, e_i and e_i are both marked in m_{i+1} because they are not incoming edges of n.

2. $e_i \in \circ n$ $e'_i \notin \circ n$

   By the definition of path, as e'_i is in p, the target n is in p and at least one of its outgoing edge e_{i+1} is in p. By the workflow graph semantics, there exists a transition T such that $m_i \xrightarrow{T} m_{i+1}$ and e'_{i+1} is marked in m_{i+1}.

   By the workflow graph semantics, e'_i is marked in m_{i+1} because e'_i is not an incoming edge of n.

   Thus, m_{i+1} is in M.

3. $e_i \notin \circ n$ $e'_i \in \circ n$

   A similar reasoning as for the case $e_i \in \circ n$ $e'_i \notin \circ n$ can be applied to show that there exists a transition T and a marking $m_{i+1} \in M$ such that $m_i \xrightarrow{T} m_{i+1}$.

Because W is acyclic, M is a finite set and we cannot reach the same marking twice. Therefore, a marking m_t such that $m_t[e_{n-1}] = 1$ or $m_t[e'_{n-1}] = 1$ is reachable.

By the definition of workflow graph semantics: In marking m_t, executing n_n once results in a marking m'_t where either e_{n-1} or e_k are marked. Thus n_n can also be executed in m'_t. Executing n_n twice results in the outgoing edge of n_n being marked with two tokens. Thus exists a reachable state of W where an edge is marked with more than one token, i.e., there is a lack of synchronization in W. □

The outline of the 'only if' direction of the proof of Thm. 5.16 is that, whenever there is a handle, this handle can be 'executed' in the sense that there exists an execution such that a token reaches the incoming edge of the start node of the handle and then two tokens can be propagated to reach two incoming edges of the end node of the handle to create a lack of synchronization. We believe that, due to its direct relationship with an

erroneous execution, the handle is an adequate error message for the process modeler. In Fig. 5.5, the handle corresponding to the lack of synchronization is highlighted. We say that the end node of the handle is the *location* of the lack of synchronization. Note that it is necessary that the workflow graph is deadlock-free in order to show that the handle can be executed and thus a lack of synchronization be observed. However, even if the workflow graph contains a deadlock, a handle is a design error because, once the deadlock is fixed, the handle can be executed and a lack of synchronization can be observed.

5.2.2. Detecting handles

Given an acyclic directed graph $G = (N, E)$ and four different nodes $s_1, s_2, t_1, t_2 \in N$, Perl and Shiloach (59) show how to detect two node-disjoint paths from s_1 to t_1 and from s_2 to t_2 in $O(|N| \cdot |E|)$. We extend their algorithm in order to detect two edge-disjoint paths between two nodes of an acyclic workflow graph.

Our intuitive view of the approach proposed by Perl and Shiloach (59) is to try to move two tokens from a start node to an end node of a handle without allowing the two tokens to visit the same edge, i.e., without allowing the tokens to follow paths that overlap. To achieve this, we build a so-called *state graph* which records the state space of the possible combinations of marked edges and a transition relation. Note that the size of the state graph is quadratic with respect to the number of edges of the original graph because we only consider combinations of two edges. Checking the existence of a handle, i.e., two edge-disjoint paths between some particular types of nodes, in the workflow graph reduces to checking the existence of a special path in the state graph.

The key property of this special path in the state graph is that it does allow the two tokens to follow paths that overlap in the original workflow graph. This is prevented by restricting the state graph in two ways:

1. The state graph does not contain the states representing two tokens are on the same edge of the original workflow graph.

2. The transition relation of the state graph is restricted in a way that, if two tokens were to visit paths that overlap in the original workflow graph, they would mark the overlapping edge at the same time (which is ruled out by 1). This synchronization is achieved by only allowing the token that is the most upstream in the original workflow graph to move.

In the remainder of the this section, we describe how to obtain a *line graph*: a directed graph in which the edges are represented as nodes. Based on the line graph and a numbering allowing us to detect which edge is upstream from another, we show how to build the state graph. Finally, we describe how to detect handles using the state graph.

5.2.2.1. Computing a line graph

Perl and Shiloach (59) describe how to detect two node-disjoint paths in a directed graph whereas we want to detect two edge-disjoint paths in a workflow graph which is a directed multi-graph. To do so, we transform the workflow graph into its *line graph*.

A line graph G' of a graph G represents the adjacency between edges of G. Each edge of G becomes a node of G'. Additionally, we carry over those nodes from G to G' that can be the start or end nodes of a handle, i.e., AND-splits and IOR-splits or XOR-joins, respectively. The edges of G' are such that the adjacency in G is reflected in G'. For the workflow graph in Fig. 5.5, we obtain the line graph shown in Fig. 5.6. The line graph has two node-disjoint paths from an AND- or IOR-split to an XOR-join if and only if the workflow graph has a handle from that split to that join.

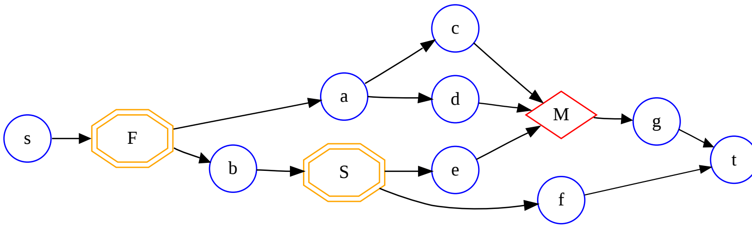


Figure 5.6.: The line graph of Fig. 2.4.

In the following, for any edge e of a directed graph $t(e)$ references the target node of e and $s(e)$ references the source node of e.

Definition 5.17 (Line graph). *Let $W = (N, E, c, l)$ be an acyclic workflow graph. The* line graph *$L = (N', E')$ of W is a directed graph such that:*

$$
\begin{aligned}
S &= \{x \mid x \in N \wedge x \text{ is an AND-split or an IOR-split}\} \\
T &= \{x \mid x \in N \wedge x \text{ is an XOR-join}\} \\
N' &= E \cup S \cup T \\
E' &= \{a \to b \mid a, b \in E \wedge t(a) = s(b)\} \\
&\cup \{a \to b \mid a, b \in N' \wedge (\ (a \in S \cup T \wedge a = s(b)) \vee (b \in S \cup T \wedge t(a) = b)\ )\}
\end{aligned}
$$

Fig. 5.6 shows the line graph of the workflow graph of Fig. 2.4. The orange octagons with double line are nodes in S. The red diamonds are the nodes in T.

5.2.2.2. Generating the state graph

We build a *state graph* from the line graph of an acyclic workflow graph. Each node of the state graph is a multi-set containing two nodes of the line graph, i.e., two edges of the original workflow graph. Such a multi-set represents a *state* of the line graph where

the two nodes carry a token. The edges of the state graph represent the allowed token moves. In order to determine which node of the line graph is upstream of another, i.e., to determine the allowed token moves, we compute a value $v(n)$ given by the reverse post order numbering of the nodes (33). The token on the node with the lowest reverse post order value is allowed to move. Similarly to Yang *et al.* (84), we perform a straightforward extension of the algorithm of Pearl and Shiloach (59): We add to the state graph the states where there are two tokens on a node in S or two tokens on a node in T. This allows us to check for two disjoint paths between one pair of node in S and T instead of two pairs of nodes as originally described by Perl and Shiloach.

Definition 5.18 (State graph). *A* state graph *of an acyclic simple directed graph* $L = (N', E')$ *is an acyclic directed graph* $F = (N'', E'')$ *such that:*

$$
\begin{aligned}
N'' &= \{[x,y] \mid x,y \in N \wedge x \neq y\} \cup \{[x,x] \mid x \in S \cup T\} \\
E'' &= \{[x,y] \to [x',y] \mid x \to x' \in E' \wedge v(x) \leq v(y)\}
\end{aligned}
$$

Fig. 5.7 shows the state graph of the line graph of Fig. 5.6. As shown by Perl and Shiloach (59) the number of edges $|E''|$ in the state graph is in $O(|N| \cdot |E|)$ in terms of the original graph.

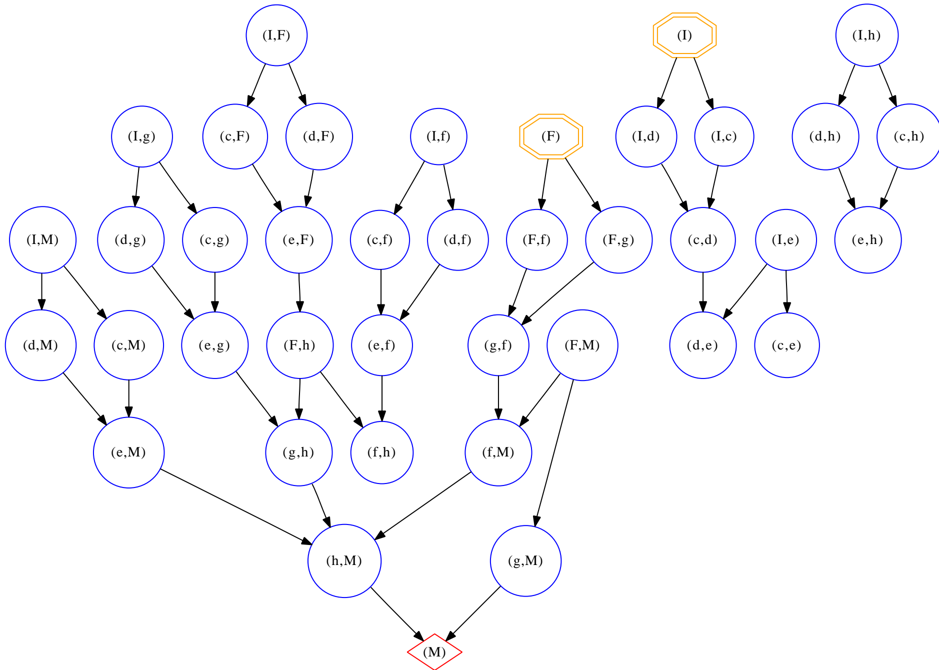


Figure 5.7.: The state graph of Fig. 5.6. .

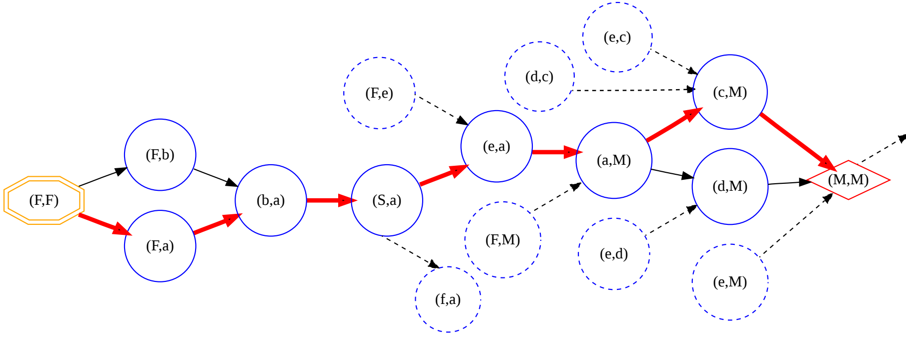


Figure 5.8.: A portion of the state graph for the line graph in Fig. 5.6.

5.2.2.3. Checking for handles

To detect a handle in the original workflow graph, we have to check if there exists a path in the state graph from a state with two tokens on a node in S to a state with two tokens on a node in T. This is achieved by traversing the graph from each unvisited node in S until reaching a node in $t \in T$ or having visited all the reachable nodes. In the worst case, the graph traversal visits each node of the graph once. The edges that belong to the handle in the workflow graph can be easily retrieved: They are the edges that are on the path from the state with two tokens on a node $s \in S$ to the state with two tokens on a node $t \in T$ of the handle in the state graph. On Fig. 5.7 we observe that there is a path between $[I, I]$ and $[O, O]$ which indicates that there is an handle between I and O in workflow graph of Fig. 2.4.

To decide whether there are such two node-disjoint paths in the line graph, we can now apply the approach by Perl and Shiloach (59), which is the construction of a graph that we call the *state graph*. To this end, we extend the partial ordering $<$ of the nodes in the line graph to a total ordering $\prec$. A node of the state graph is a pair (n, m) of nodes of the line graph such that either $n = m \in S \cup T$ or $n \neq m$ and $n \prec m$. There is an edge in the state graph from (n, m) to (n', m) (or to (m, n')) if there is an edge from n to n' in the line graph.

Figure 5.8 depicts a portion of the state graph for the line graph in Fig. 5.6. We have two node-disjoint paths from an AND- or IOR-split s to an XOR-join j in the line graph if and only if there is a path from (s, s) to (j, j) in the state graph. In Fig. 5.8, one such path is highlighted which indicates two disjoint paths from the AND-split F to the XOR-join M. The number of edges in the state graph is in $O(|N| \cdot |E|)$ and the number of nodes is in $O(|N|^2)$ in terms of the line graph (59). The entire algorithm can be implemented to run in quadratic time in the size of the workflow graph, cf. (21).

5.2.3. Combining symbolic execution with handle detection

Symbolic execution detects deadlocks in a prefix of the workflow graph that is free of lack of synchronization. Therefore, we first check the workflow graph for handles. We use the end nodes of the handles to delimit a maximum prefix of the workflow graph that is free of handles. We perform a symbolic execution of this prefix. If a deadlock is detected, we report the deadlock. If symbolic execution labels the incoming edges of the end node of a handle, we report the corresponding lack of synchronization. If no deadlock is detected and there is no handle detected, the workflow graph is sound.

5.2.4. Sometimes-concurrent

We now discuss the notion of sometimes-concurrent edges:

Definition 5.19 (Sometimes-concurrent, mutually exclusive)**.**

- *Two edges are* sometimes-concurrent *if there exists an execution in which they are parallel.*
- *Two edges are* mutually-exclusive *or* never-concurrent *if they are not sometimes-concurrent.*

The notion of sometimes-concurrent edges is tightly related to lack of synchronization: It follows from the proof of Thm. 5.16 that two incoming edges e, e' of an XOR-join are sometimes-concurrent if and only if there is handle to this XOR-join such that one path goes through e and the other goes through e'.

Beyond checking for lack of synchronization knowing whether two elements are sometimes-concurrent has various use cases: For example, some modeling languages support so-called termination nodes, which shutdown the execution of a process regardless of the status of any concurrent activity. As part of the control-flow analysis in an industrial tool, it was necessary to ensure that termination nodes are never executed concurrently, i.e., that termination nodes are mutually-exclusive, in order to prevent unexpected behaviors at runtime. It is also useful for data-flow analysis: A data-flow hazard may arise if two conflicting operations on the same data object are executed concurrently. This can happen only if the tasks containing the data operations are sometimes-concurrent. A task of a process is represented as an edge in the corresponding workflow graph. Thus for the purpose of data-flow analysis, we are interested in detecting sometimes-concurrent elements.

To decide whether two arbitrary edges of a sound graph are sometimes-concurrent, we show the following:

Lemma 5.20. *In a sound prefix of the workflow graph W, if two edges e_1, e_2 are sometimes-concurrent, then $e_1 \mid\mid e_2$.*

Proof. This lemma can be proven by contradiction: Without loss of generality, assume that $e_1 < e_2$. As e_1 and e_2 are sometimes-concurrent, there exists a reachable marking m such that $m[e_1] = m[e_2] = 1$. As there is no deadlock, we can move the token on e_1 on the path to e_2 until reaching a marking m' such that $m'[e_2] = 2$. The marking m' is a lack of synchronization which is ruled out by the soundness assumption. □

We can now determine whether two edges, or elements, are sometimes-concurrent:

Theorem 5.21. *It can be decided in quadratic time with respect to the size of the workflow graph whether a given pair of edges is sometimes-concurrent.*

Proof. By Lemma 5.20, $e_1 \mid\mid e2$ or both edges are not sometimes concurrent. In the case $e_1 \mid\mid e2$, let W^* be the graph obtained by removing all the elements of the workflow graph that follow either e_1 or e_2 and add an XOR-join x to be the target of e_1 and e_2. The edges e_1 and e_2 are sometimes-concurrent if and only if x is the end node of a handle in W^*. We have shown in the previous section how to check for such an handle in quadratic time. □

5.3. Dealing with over-approximation

In this section, we show how the labeling that is computed in the symbolic execution can be leveraged to deal with errors that are detected in the workflow graph but may not arise in a real execution of the process due to the correlation of data-based decisions.

5.3.1. User interaction to deal with over-approximation

When we capture the control-flow of a process in a workflow graph, we abstract from the data-based conditions that are evaluated when executing an XOR-split or an IOR-split of the process. Such a data-based decision can be, for example, isGoldCustomer(*client*). The data-abstraction may result in errors that occur in the workflow graph but not in an *actual* execution of the process. We use in the following the term *actual execution* to refer to an execution of the real process as opposed to its workflow graph, which is an abstraction of the process.

For example, the graph in Fig. 5.9 contains a deadlock located at J. However, if the data-based decisions in all actual executions are such that outcome d is taken whenever e is taken, this deadlock would never occur in an actual execution. For example, the data-based condition on d could be exactly the same as on

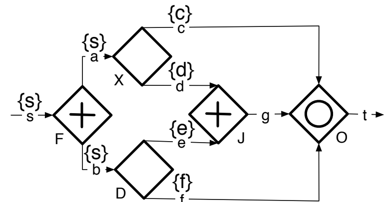


Figure 5.9.: A deadlock.

e. The user should therefore have the opportunity to inspect the deadlock and decide whether outcomes d and e are related as mentioned above and then dismiss the deadlock. Analysis of the graph should then continue.

To inspect a deadlock, we provide the AND-join, two incoming edges e, e' of the join, and their non-equivalent labels $\varphi(e), \varphi(e')$ to the user. Then, she has to decide whether for each outcome $o \in \varphi(e)$ and each actual execution where o is taken, there is an outcome $o' \in \varphi(e')$ that is also taken in that execution and vice versa. If the user affirms the latter, she can dismiss the deadlock. This basically postulates the equivalence of the two symbols in actual executions. Henceforth, we continue the symbolic execution by treating, internally to the analysis, the AND-join as an IOR-join.

To inspect a lack of synchronization, we provide the XOR-join that terminates the detected handle and the two incoming edges e, e' of the XOR-join that are part of the handle to the user. Furthermore, we provide the labels $\varphi(e), \varphi(e')$. Then, the user has to determine that for each pair of outcomes $o \in \varphi(e)$ and $o' \in \varphi(e')$, we have that o is taken in an actual execution implies that o' is not taken in that execution. If the user affirms the latter, she can dismiss the lack of synchronization. This basically postulates that o and o' are mutually-exclusive in actual executions. If this is done for all incoming edges of the XOR-join, we can henceforth continue the symbolic execution by treating, internally to the analysis, the XOR-join as an IOR-join. Figure 5.10 shows an example with a lack of synchronization located at J. The user may dismiss it because for example, the conditions on c and e are the same, i.e., d and e are mutually-exclusive.

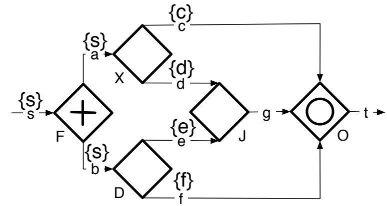


Figure 5.10.: A lack of synchronization.

Figure 5.11 shows another example where the deadlock can be dismissed if b and c are deemed to be equivalent. Once the user dismissed the deadlock, we continue the symbolic execution and label the edge d with the symbol $\{b, c\}$ according to the IOR-join propagation rule. To dismiss the lack of synchronization at M, the user then has to check the pair a, b and the pair a, c for mutual exclusion.

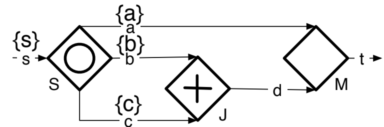


Figure 5.11.: A deadlock and a lack of synchronization.

The deadlock displayed on Fig. 5.4, can be dismissed if g is equivalent to s, i.e., g is deemed to be marked in every execution of the process.

Note that, if we provided an execution, i.e., an error trace, rather than the symbolic information to dismiss an error, we would present exponentially many executions that contain the same error in the worst case. The analysis of the outcome sets precisely gives the conditions under which one deadlock or one lack of synchronization occurs. It does not contain information that is irrelevant for producing the error.

5.3.2. Relaxed soundness

In some cases, the user should not be allowed to dismiss an error. Figure 5.12 shows a deadlock that cannot be avoided unless d and e are never taken which clearly indicates a modeling error. This is related to the notion of *relaxed soundness* (6). A workflow graph is *relaxed sound* if for every edge e, there is a *sound execution* that marks e, where an execution is *sound* if it neither reaches a deadlock nor a lack of synchronization.

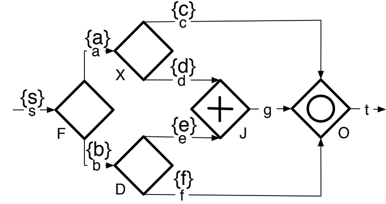


Figure 5.12.: A deadlock located at J that should not be dismissed.

The graph in Fig. 5.12 is not relaxed sound. We do not know any polynomial-time algorithm to decide relaxed soundness for acyclic workflow graphs. However, we provide here necessary conditions for relaxed soundness that can be checked in polynomial time.

One necessary condition for relaxed soundness is that for every AND-join j and every pair of incoming edges e, e' of j, e and e' are sometimes-concurrent. Likewise, for every XOR-join j and every pair of incoming edges e, e' of j, e and e' must not be always-concurrent. Moreover, we have the following stronger necessary conditions:

Theorem 5.22. *Let W be an acyclic workflow graph.*

1. *If for an AND-join j, and a pair of incoming edges e, e' of j and one outcome $o \in \varphi(e)$, we have that all outcomes in $\varphi(e')$ are mutually-exclusive with o, then W is not relaxed sound.*

2. *If for an XOR-join j, and a pair of incoming edges e, e' of j, we have $\overline{\varphi}(e) \cap \overline{\varphi}(e') \neq \emptyset$, then W is not relaxed sound.*

Proof.

1. Assume that there exists an AND-join j, a pair of incoming edges e, e' of j, and one outcome $o \in \varphi(e)$, such that every outcome $o' \in \varphi(e')$ is mutually exclusive with o. When o carries a token during an execution σ, then e carries a token during σ by Lemma 5.6. As the outcomes in $\varphi(e')$ are mutually exclusive with o, e' is not marked during σ. By Proposition 5.4, there is a deadlock located at j during σ. Thus, there is no sound execution that marks o, i.e., W is not relaxed sound.

2. Assume that for an XOR-join j, and a pair of incoming edges e, e' of j, we have $\overline{\varphi}(e) \cap \overline{\varphi}(e') \neq \emptyset$. Then, there exists an outcome $o \in \overline{\varphi}(e) \cap \overline{\varphi}(e')$. By Lemma 5.6, when o is marked during an execution σ, e and e' get marked during σ and therefore ther exists and execution σ' that is interleaving equivalent to σ and leads to a lack of synchronization. Thus, there exists no sound execution that marks o, i.e., W is not relaxed sound.

□

Based on the previous results in this chapter, we can compute these necessary conditions for relaxed soundness in polynomial time. If one of them is true, the corresponding error should not be dismissible. For example, the deadlock in the workflow graph depicted by Fig. 5.12 cannot be dismissed because d and e are mutually-exclusive. The lack of synchronization located at J in the workflow graph depicted by Fig. 5.13 cannot be dismissed because $\overline{\varphi}(d) = \{d\}$ and $\overline{\varphi}(b) = \{s, c, d\}$ and thus $\overline{\varphi}(e) \cap \overline{\varphi}(e') \neq \emptyset$.

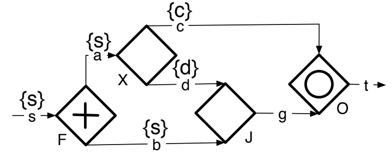


Figure 5.13.: A lack of synchronization that should not be dismissed.

Note that, deciding soundness and relaxed soundness complement each other. If we only decided relaxed soundness, we would not detect the deadlock that may be present in an actual execution of Fig. 5.9 for example.

5.4. Weak soundness is NP-Hard

In this chapter, we have shown that we can decide soundness of an acyclic workflow graph, i.e., that every execution is sound, in quadratic time and space. In Sect. 5.3.2, we gave two necessary conditions for relaxed soundness, i.e., there exists a set of sound executions such that each edge of the workflow graph is marked by at least one sound execution, and we have shown that we can compute those in polynomial-time. However, no polynomial-time decision procedure is known for relaxed soundness, even for workflow graph without IOR-joins. In this section, we complement these results by proving that deciding another notion of correctness called *weak soundness* (cf. Def. 2.26), i.e., deciding whether there is exists at least one sound execution of an acyclic workflow graph, is NP-hard:

Theorem 5.23. *Deciding whether a workflow graph is weakly sound is NP-hard.*

Proof. We prove this by giving a linear reduction of the boolean satisfiability problem, which is NP-complete [1] (4), into deciding whether an acyclic workflow graph is weakly sound.

Let a boolean formula $f = (l_1 \vee l_2 \vee l_3) \wedge (l_4 \vee ...) \wedge ...$ expressed in conjunctive normal form where each term l_i can be either a variable (x) or a negated variable ($\neg x$).

[1] We actually consider a formula in conjunctive normal form in which case the problem is still NP-complete as long as the number of variable per clause is not bounded by a number lower than 3. More precisely, the *k-SAT problem* is the problem of finding a satisfying assignment to a boolean formula expressed in conjunctive normal form such that each disjunction contains at most k variables. The *k-SAT problem* is NP-complete for any $k > 2$.

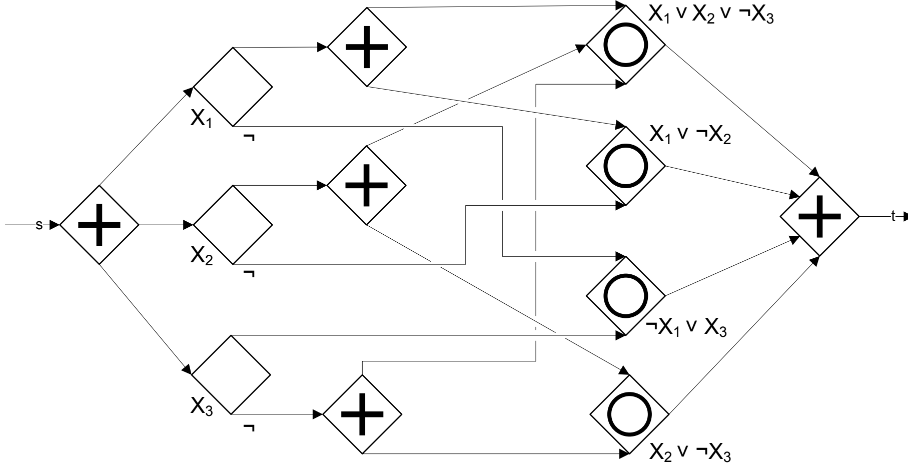


Figure 5.14.: The workflow graph obtained for the formula $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$.

Without loss of generality, we can remove any clause containing a variable which only appears negated or only appears non-negated in the formula [1].

We show how to build a workflow graph $W = (N, E, c, l)$ such that W is weakly sound if and only if f is satisfiable. Fig. 5.14 illustrates the workflow graph obtained for the formula $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$.

W is built as follows:

- The set of nodes N contains:
  - an initial AND-split.
  - an XOR-split for each variable of f,
  - an AND-split s_i for each literal l_i that appears more than once in f,
  - an IOR-join for each clause of f,
  - and a final AND-join for the conjunction,
- The set of edges E contains:
  - the source edge targeting the initial AND-split,
  - an edge from the initial AND-split to each XOR-split,
  - an edge for each literal l_i that appears more than once, from the XOR-split corresponding to the variable in l_i to the AND-split s_i that corresponds to l_i,

[1] This step only simplifies the description of the reduction. It is also possible to perform the reduction without performing this step.

- an edge e_i for each occurrence of a literal l_i in a clause c_j of f, such that the target of e_i is the IOR-join corresponding to c_j and the source of e_i is the AND-split corresponding to l_i if l_i appears multiple times in f, the source of e_i is the XOR-join corresponding to the variable represented in l_i otherwise,
- an edge from each IOR-join to the AND-join,
- and the sink edge that has the AND-join as source.

This transformation is linear with respect to the length of the formula. The number of nodes is bounded by 3 times the number of variable plus the number of clauses plus two (initial AND-split and AND-join). The number of edges is bounded by 3 times the number of variables plus the number of clauses plus the number of literals plus the additional source and sink.

Each XOR-split of W corresponds to a variable of f and has two outgoing edges that correspond to the true and false valuations of the variable, respectively. Each IOR-join of W corresponds to a clause of f. The key characteristic of W is that, for any edge e corresponding to a variable valuation, there is a path from e to an edge e_i representing a literal l_i that targets the IOR-join representing the clause containing l_i. The path is either trivial, i.e., $e = e_i$, or is composed of e, an AND-split, and e_i. By the workflow graph semantics, this property implies that *(I) e carries a token during σ if and only if e_i carries a token during σ*. Note that the clause evaluates to true when the variable takes valuation corresponding to e.

W **is weakly sound only if f is satisfiable** Assume that W is weakly sound, i.e., there exists a sound execution σ.

By construction, each XOR-split is the target outcome of the AND-split targeted by the source edge. Therefore, each XOR-split is executed during σ by the workflow graph semantics. For each XOR-split one and only one of the two outgoing edges carries a token during σ by the XOR-split semantics. Thus for any XOR-split, the outgoing edge e such that $\vec{\sigma}(e) = 1$ gives us the valuation of the variable v corresponding to the XOR-split: the valuation is true whenever e represents a true valuation of the variable, it is false whenever e represents a false valuation of the variable. We obtain the mapping $k : V \rightarrow \{true, false\}$ which maps each variable of f to a true or false valuation.

As each valuation of the variables appears in at least one clause of the formula and because of (I), some IOR-join incoming edge carry a token during σ and thus some IOR-join is executed during σ. Therefore some incoming edges of AND-join carry a token.

Because σ is a sound execution, all incoming edges of the AND-join must carry a token during σ. Therefore, at least one of the incoming edge of each IOR-join carries a token in σ.

By (I), this implies that for each clause there is variable valuation of k that makes the clause true, i.e., k is an assignment of the variable of f that make f evaluate to true.

W is weakly sound if f is satisfiable Assume that f is satisfiable, i.e., there exists a mapping $k : V \to \{true, false\}$ that maps each variable of f to a valuation such that replacing each variable by its valuation makes f evaluate to true.

We use k to define a sound execution σ of W:

σ starts in the initial marking where the source edge is marked.

Then the initial AND-split executes.

Each XOR-split x executes, we use k to choose which outcome of x carries a token: Let v be the variable corresponding to x if $k(v) = true$ then the edge corresponding to the true valuation carries a token, if $k(v) = false$ then the edge corresponding to the false valuation carries a token.

By the workflow graph semantics and because there is no following XOR- or IOR-split, the rest of the execution is fully deterministic aside from the interleaving of concurrent events.

By the satisfiability assumption, each clause of f evaluates to true when the variables are replaced according to the valuations given by k. Therefore, by (I) every IOR-join has at least one incoming edge carrying a token during σ and thus executes during σ.

Each incoming edge of the AND-join carries a token during σ and the AND-join executes during σ.

As the AND-join executes during σ it is clear that σ does not reach a deadlock. It is also clear by the format of the constructed workflow graph that σ does not reach a lack of synchronization.

□

5.5. Summary

We have shown how basic relationships between control-flow edges of a process can be decided in polynomial time for acyclic workflow graphs with inclusive OR gateways. This can be used to detect control-flow errors but could also be used in variety of other use-cases like to perform data-flow analysis, or to compare processes at a behavioral level. We presented a control-flow analysis that decides soundness in quadratic time and gives concise error information that precisely characterizes a single error. We outlined how the diagnostic information can be used to efficiently dismiss spurious

errors that may not occur in actual executions of the process due to correlated data-based decisions. We complemented these results by showing that deciding the weak soundness of a workflow graph containing IOR-join is an NP-hard problem.

5.6. Related work

While symbolic execution approaches are used in other domains, such as program verification for exmaple, that are out of the scope of this thesis, our symbolic execution is novel for the verification of workflow graph and workflow nets.

The initial inspiration for the symbolic execution came from BPEL's dead path elimination (DPE) (39). DPE is not an analysis technique but a control-flow execution strategy for acyclic process models. The idea of DPE that inspired the symbolic execution is to propagates two types of tokens: 'true' and 'false' tokens. Such an execution strategy allows, for example, the IOR-join enablement condition to be transformed into a local condition: the IOR-join is enabled if and only if all of its incoming edges carry a token. Executing the IOR-join results in a true token if at least one of the incoming edges carries a true token, a false token otherwise. During the development of the symbolic execution we transformed the idea to propagate information in the tokens, even when a path is not executed (false tokens), into propagating the conditions under which the current edge carries a token.

Eshuis and Kumar (11) present an approach based on integer programming (IP) to check for control-flow errors. They encode a workflow graph into an IP model. Finding a solution to this IP model implies that the workflow net is weakly sound. Then propose to check for a solution of a tweaked IP model for each join to establish soundness of the model. Similarly to our work combine with structural decomposition approach, they can only deal with acyclic processes and processes containing well structured loops. The approach can also delivers an error trace as diagnostic information and points at the joins where the error occurs. The approach has an exponential time complexity with respect to the size of the input model. This exponential time complexity is also observed in practice on the experiments that they carry.

As pointed out during the presentation the notion of handles relates to the notion of T/P handles and therefore our notion of handles is similar to the one of Esparza and Silva (14) for Petri nets. If we restrict ourselves to workflow graphs without IOR gateways, one of the directions of our characterization follows from the result of Esparza and Silva. The converse direction does not directly follow. The structural soundness characterization of Esparza and Silva that uses the notion of handles applies to free-choice Petri net in general, including cycles, but does not apply to IOR-join. There is also no efficient way that is known to check this characterization. Our notion of handles has been described by van der Aalst (66) for workflow net. This work was discussed already in Sect. 3.7. In summary this work shows a sufficient but not necessary condition to structural soundness and an approach to compute it for a, possibly cyclic,

workflow. Van der Aalst also points out that these handle can be computed using maximum flow approach. Given the state of the art in computing the maximum flow (cf. a survey from Goldberg and Tarjan (31)), the complexity of detecting handles with such an approach is at best $O(|N|^3 \cdot |E| \cdot log(|N|)$.

On the control-flow relationships side, Kovalyov and Esparza (43) describe a technique to identify, in cubic time, sometimes-concurrent elements in live and bounded free-choice net and thus, by Thm. 2.25 and the translation presented in Sect. 2.4, in a sound workflow graphs that do not contain IOR-logic.

As already pointed out, numerous (non-symbolic) approaches to check for control-flow errors exists, we discussed the structural approaches in Sect. 3.7 and will discuss the approaches based on state space exploration in Sect. 6.7.

Part III.

Analysis of Industrial Business Process Models

6

Building an Analyzer

The control-flow analysis techniques presented in the earlier chapters combine a polynomial time worst-case complexity and provide some valuable diagnostic information to the modeler. However, their individual coverage is limited: the structural analysis cannot deal with processes containing IOR-joins and the symbolic execution can only deal with acyclic process models.

In this chapter, we show how we integrate our techniques to build an analyzer that overcomes these individual coverage limitations. This analyzer aims to satisfy the three requirements discussed in Sect. 1.2.2 that are: consumability, efficiency, and coverage. To achieve this goal, we built a hybrid analyzer which leverages the advantages of the structural analysis approach presented in Chap. 3 and symbolic execution combined with the handle checking technique presented in Chap. 5 in terms of consumability and efficiency. Additionally, we use other state-of-the-art approaches to ensure coverage of business process models containing cycles and IOR-joins.

In Sect. 6.1, we motivate our choice to use workflow graphs to model the control-flow of business process models and briefly discuss the existing translations allowing to model these control-flows by workflow graphs. In Sect. 6.2, we present additional techniques from the state of the art that we use in our analyzer. In Sect. 6.3, we present the architecture of our analyzer, i.e., the way we combine the different techniques to build our analyzer. In Sect. 6.5, we present the user interaction that it provides, i.e., the diagnostic information that the techniques deliver, the fix suggestions that can be inferred from the diagnostic information, and the capabilities to dismiss some errors. In Sect. 6.6, we summarize the different versions of this analyzer that have been integrated into IBM products.

6.1. Modeling the control-flow of a business process

We use workflow graphs to model the control-flow of business processes. This abstraction of the original modeling language has two main motivations:

1. It provides an extra level of abstraction and allows us to decouple our analyzer from the original modeling language of the process. This allows us to apply our verifications techniques to business process models expressed in various languages such as BPMN 2.0 (54), BPEL (39), or IBM WebSphere Business Modeler language by replacing only the component translating the process model control-flow into a workflow graph.

2. Additionally, it allows us to cope with the complexity of the original modeling languages by translating complex and language specific features into workflow graph patterns. For example, various languages support the modeling of activities with multiple incoming and outgoing edges. The control-flow logic of such an activity is usually specified as a property of the activity or as a properties of its adjacent edges. The same semantics can be modeled by a task, with one incoming and outgoing edges, and a composition of gateways connecting to the input and output edges (20, 25).

During the course of this work we developed many translations to workflow graphs including translations from BPMN, BPEL, and the IBM WebSphere Business Modeler language. Describing such a translation is tedious and out of the scope of this thesis. Translations from IBM WebSphere Business Modeler language are described in other publications (18, 19, 20). A technical report (25) describes the translation from BPMN models to workflow graphs. The translation from BPEL to workflow graphs is not documented but is also the most direct of the three. Translations from BPEL processes into Petri-net have been described by other authors (47, 55, 63).

6.2. Additional techniques

As discussed earlier, we build a hybrid analyzer. There are four techniques from the literature that we use in this analyzer that were not yet presented in this thesis:

1. In Sect. 6.2.1, we present the *refined process structure tree (RPST)* which allows us to decompose a workflow graph in smaller fragments that can be analyzed in isolation.

2. In Sect. 6.2.2, we present heuristics that allow us to efficiently sort out some sound fragments that have a simple but common structure in order to speed up the analysis.

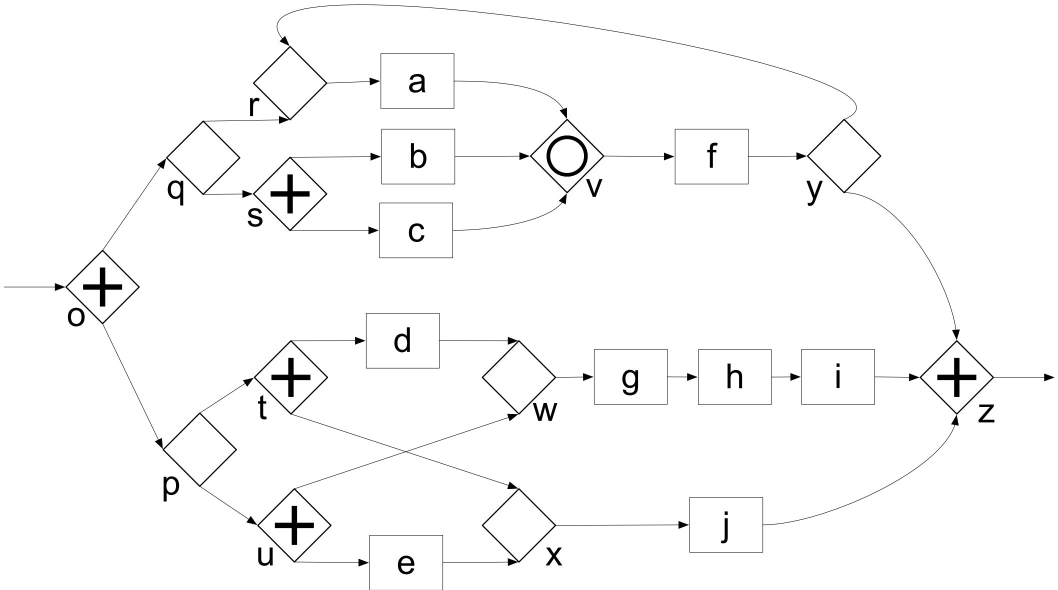


Figure 6.1.: An example of workflow graph.

3. In Sect. 6.2.3, we describe a technique to complete a workflow graph with multiple sinks to obtain a workflow graph with a unique sink. This completion is necessary to be able to apply the RPST and the structural analysis.

4. In Sect. 6.2.4, we present the state space analysis approach which allows us to check the soundness of the fragments that cannot be checked using other techniques, i.e., cyclic fragments containing an IOR-join whose structure does not match any heuristic.

6.2.1. The refined process structure tree

We use a parsing technique called the *Refined Process Structure Tree (RPST)* (73). The RPST decomposes a workflow graph into a hierarchy of *fragments* with a single entry and single exit (SESE) of control. A SESE fragment of a workflow graph is a subgraph that has a single entry node and a single exit node. Figure 6.2 illustrates the result of applying the decomposition on the workflow graph illustrated by Fig. 6.1, which is decomposed into such fragments using the RPST parsing.

Note that the RPST Algorithm splits some gateways into multiple nodes. For example, the IOR-join v in Fig. 6.1 is split into v' and v" in Fig. 6.2 by the RPST Algorithm. This does not come from the original description of RPST computation but is a nor-

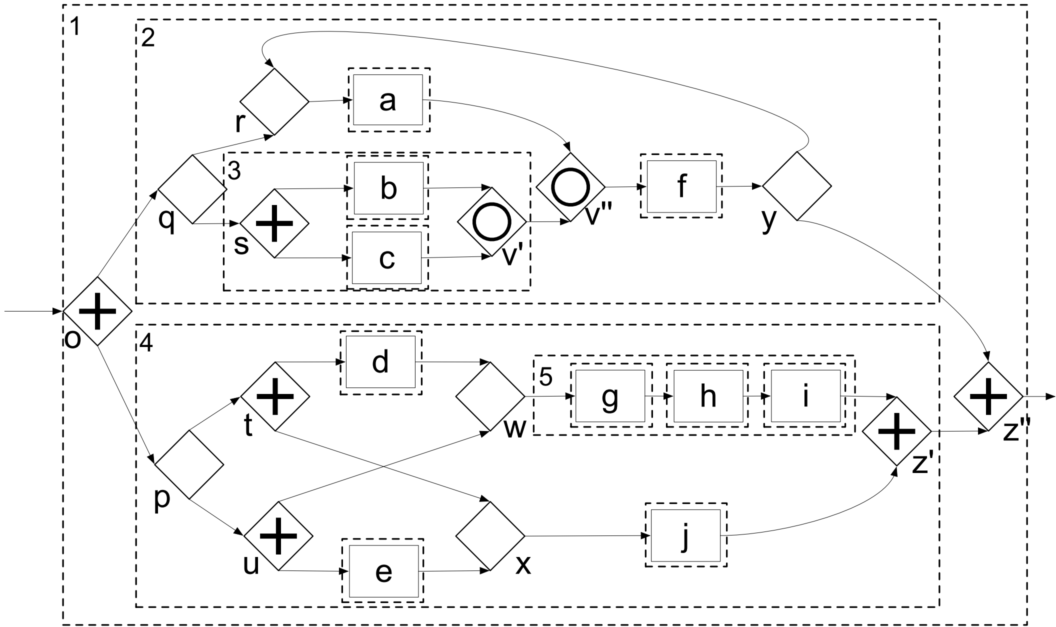


Figure 6.2.: Decomposition of the workflow graph of Fig. 6.1 using the Refined Process Structure Tree parsing.

malization step described elsewhere (72)[1] that we adopt to ensure that fragments have a unique entry edge and a unique exit edge which will become handy when analyzing fragments in isolation because now each fragment has a single source edge and a single sink edge.

Formally, a fragment is defined as follows:

Definition 6.1 (Fragment). *Let $W = (N, E, c, l)$ be a workflow graph. A* fragment *$F = (N', E', c', l')$ is a sub-graph of W such that:*

- *F is non-empty*
- *$N' \subseteq N$ and $E' \subseteq E$*
- *$\exists e, e' \in E$ with $E \cap ((N \setminus N') \times N') = \{e\}$ and $E \cap (N \times (N \setminus N')) = \{e'\}$*

We call e the entry edge *of the fragment and e' the* exit edge *of the fragment.*

[1]Vanhatalo describes two decomposition approaches, the *Normal Process Structure Tree (NPST)* and *Refined Process Structure Tree (RPST)* and a normalization approach to turn a RPST into a NPST by splitting nodes that are the on boundary of a RPST fragment. We use the RPST decomposition and the normalization step to deal with fragments that are in the form of NPST fragments.

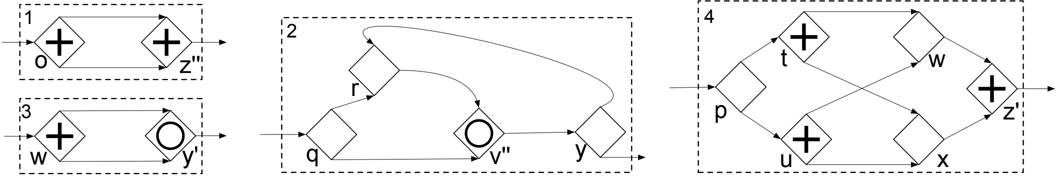


Figure 6.3.: The most complex fragments of Fig. 6.2 that can be analyzed in isolation.

The fragments computed by the RPST are either disjoint or nested and there is no node n that belongs to two different fragments, i.e., the boundaries of the fragments never overlap. Therefore, the fragments can be organized as a tree that is called *refined process structure tree (RPST).*

When the workflow graph does not contain IOR-joins, soundness is compositional with respect to SESE fragments, i.e., each fragment can be checked in isolation (74).

Theorem 6.2 (Compositional soundness (72, 74))**.** *A workflow graph is sound if and only if all its child fragments are sound and the workflow graph that is obtained by replacing each child fragment with an edge is sound.*

To verify the soundness of a fragment, each of its child fragments can be treated as an edge of the workflow graph. Figure 6.3 illustrates the fragments of Fig. 6.2 in isolation.

Note that Thm. 6.2 can be extended to workflow graphs containing IOR-joins with certain restrictions on cyclic fragments containing IOR-joins (80). When decomposing a workflow graph containing IOR-joins, our implementation only decomposes the portion of the workflow graph that satisfies these restrictions.

6.2.2. Heuristics

In practice, many fragments have a simple structure that can be recognized as sound or unsound in linear time using structural heuristics (74). For example, if a fragment contains only XOR-gateways, it is purely sequential and therefore sound. In our analyzer we use a subset of the heuristics proposed by Vanhatalo (72) in his PhD thesis. We call this subset *positive heuristics* because they are the subset of the heuristics allowing us to recognize sound fragments. We deliberately choose not to use the other heuristics, which would allow us to recognize some of the unsound fragment, because they would not provide us the same kind of diagnostic information than our other analysis techniques.

The positive heuristics are derived from the following observations:

1. If there is no gateway in a fragment or all the gateways, i.e., the splits and joins, of a fragment have XOR logic, then only one path of the fragment is executed. Therefore no lack of synchronization can be reached because there is only one

token that gets propagated. No deadlock can occur because the fragment contains no AND-join or IOR-join.

2. If all gateways of a fragment have AND logic and the fragment is acyclic, then all paths of the fragment are executed concurrently. Therefore no deadlock can occur because all the edges will be provided with a token. No lack of synchronization can be reached because the fragment does not contain any XOR-join.

3. If all gateways of a fragment are either IOR-splits or IOR-joins and there exists no cycle, there cannot be a deadlock or a lack of synchronization because the execution semantics of the different IOR-joins will adapt to the output of the IOR-splits: an IOR-join will wait for all tokens produced by its preceding gateways but not for the tokens that cannot come. Since the fragment is acyclic, IOR-joins cannot introduce a deadlock.

4. In 1, replacing an AND-join by an IOR-join cannot create deadlock or a lack of synchronization. In 2, replacing a merge by an IOR-join cannot create a deadlock or a lack of synchronization.

Formally, we use these heuristics to categorize the following fragments:

Definition 6.3 (Fragment type)**.** *Let F be a fragment of a workflow graph.*

1. *F is* well-structured *if it satisfies one of the following conditions:*
   - *F has no gateway child in the process structure tree. (*sequence*)*
   - *F has exactly one XOR-split, exactly one XOR-join, no other gateway as child, the entry edge of F is the incoming edge of the XOR-split, and the exit edge of F is the outgoing edge of the XOR-join. (*alternative branching*)*
   - *F has exactly one XOR-split, exactly one XOR-join, no other gateway as child, the entry edge of F is an incoming edge of the XOR-join, and the exit edge of F is an outgoing edge of XOR-split. (*cycle*)*
   - *F has exactly one AND-split, exactly one AND-join, no other gateway as child, the entry edge is the incoming edge of the AND-split, and the exit edge is the outgoing edge of the AND-join. (*parallel branching*)*
   - *F has exactly one IOR-join, exactly one IOR-split, no other gateway as child, the entry edge is the incoming edge of the IOR-split, and the exit edge is the outgoing edge of the IOR-join. (*inclusive branching*)*
2. *F is* unstructured parallel *if it is not well-structured, contains no cycles and has no gateway with XOR-logic or IOR-split as child.*
3. *F is* unstructured alternative *if it is not well-structured and has no gateway with AND logic or IOR-split as child.*

4. *F is* unstructured inclusive *if it is none of the above, contains no cycles, and has no AND-join or XOR-join as child.*

5. *F is* uncategorized *if it is none of the above.*

Note that these fragment types are solely based on the number of gateways of each type in the fragment and the information whether the fragment contains a cycle or not. This information can be obtained in linear time with respect to the size of the workflow graph.

The fragments of the first four types described above are always sound:

Theorem 6.4 (Positive heuristics)**.** *Let F be a workflow graph fragment. The fragment F is* sound *if:*

- *F is well-structured,*
- *F is unstructured parallel,*
- *F is unstructured alternative, or*
- *F is unstructured inclusive.*

A proof of these heuristics as well as additional heuristics which identify unsound fragments are provided in the PhD thesis of Vanhatalo (72).

Getting back to the fragments of Fig. 6.2, all the fragments, aside from fragment 4, can be identified as sound using heuristics: The fragments containing only a task, or a sequence of tasks (fragment 5), are identified as well-formed sequences. The fragment 1 is well formed parallel branching. The fragment 2 is unstructured alternative. The fragment 3 is unstructured parallel. As discussed the fragment 4 is uncategorized and we will need to use another analysis technique, such as symbolic execution (cf. Chap. 5, to determine its soundness.

6.2.3. Kiepuszewski completion

A business process model might have multiple end events which are translated into multiple sinks of the corresponding workflow graph such as the workflow graph of Fig. 6.4. The computation of the RPST and the structural analysis presented in Chap. 3 require the workflow graph to have a unique sink.

To obtain a workflow graph with a unique sink, we use a completion technique proposed originally by Kiepuszewski et al. (42, Proof of Theorem 5.1) to complete a free-choice workflow net. By virtue of the results in Sect. 2.4 this technique can easily be transferred to workflow graphs. An informal description of this completion for workflow graphs was given in Sect. 4.5.2 or, more precisely, using Alg. 5 considering the sinks as the incoming edges of an IOR-join and that the dominator frontier contains only the source edge. Fig. 6.5 shows an example of a workflow graph completion using this technique.

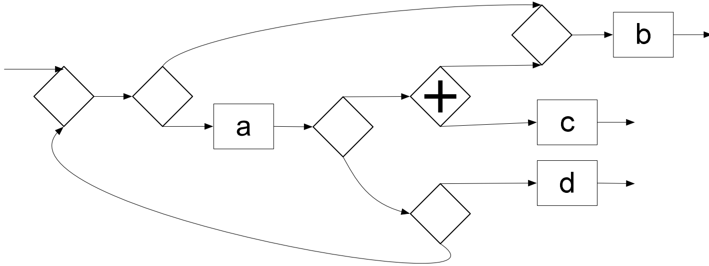


Figure 6.4.: A worklfow graph with multiple sinks.

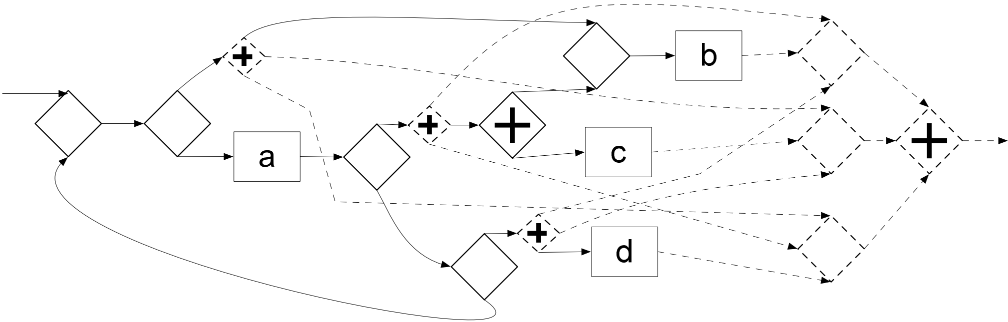


Figure 6.5.: The worklfow graph of Fig. 6.4 completed to obtain a single sink.

6.2.4. State space exploration

Using the techniques presented so far, our analyzer would not be able to deal with cyclic fragments containing an IOR-join. While good modeling practices suggest to use IOR-joins in acyclic fragments only and some languages, such as BPEL (39), even enforce this convention, other modeling languages, such as BPMN (54), allow the user to model such fragments.

To allow our analyzer to deal with cyclic fragments containing IOR-joins, we integrate a *state space exploration* module. The soundness of the fragment is decided by checking that no error marking is reachable (cf. the third characterization of soundness of Thm. 2.25) by an exhaustive search of the state space of the fragment. The state space of the fragment is explored by depth-first search. The analysis terminates upon reaching an error state and then returns a trace leading to this state. If there is no error, the entire state space is explored.

We have shown in previous work (18, 19, 20) that, when combined with RPST decomposition and heuristics, state space exploration is fast enough to analyze many industrial business process models. Moreover, state space exploration is extremely flexible, i.e., it can be extended to other modeling constructs easily. However, it remains our last resort technique for two reasons:

1. State space exploration only returns an error trace as diagnostic information, which we find less consumable than the diagnostic information produced by the other techniques.

2. State space exploration suffers from a worst-case exponential time and space complexity because the number of reachable states to explore can be exponential in the size of the workflow graph.

6.3. Architecture

Fig. 6.6 illustrates the architecture of our analyzer. First the control-flow of a business process model is translated into a workflow graph. The workflow graph is then decomposed, using the RPST technique (Sect. 6.2.1), into fragments. Each fragment is then analyzed in isolation: First, we use the heuristics described in Sect. 6.2.2, to sort out the fragments that can be identified as sound. It remains a portion of fragments of which the soundness is unknown. Then, if the fragment does not contain IOR-joins, we use the structural analysis technique described in Chap. 3 to check the soundness of the fragment. When the fragment does contain an IOR-join, it is analyzed either using the symbolic execution and the handle checking approach described in Chap. 5 if it is acyclic, or we resort to state space exploration when it is cyclic.

The preprocessing as well as the decomposition into fragments and the heuristic have linear worst-case time and space complexity. In case a fragment is identified as sound by the heuristics, it is done in linear time and space. The structural analysis

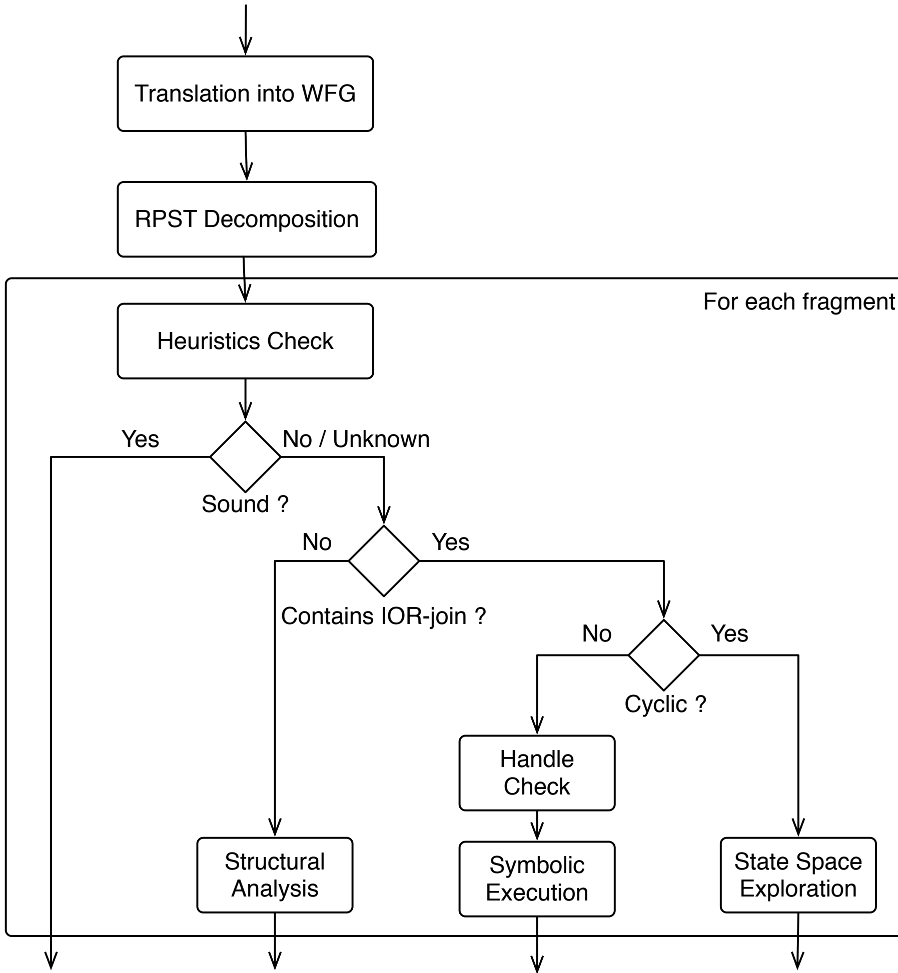


Figure 6.6.: Architecture of our control-flow analyzer.

has a polynomial time and space worst case complexity. The symbolic execution and handle checking have a quadratic worst-case time and space complexity. The state space exploration has an exponential worst case time and space complexity. The overall worst-case complexity remains exponential due to the possible use of state space exploration to analyze a cyclic fragments containing an IOR-join. We will discuss in Chap. 7, how frequently are the different techniques used when analyzing our libraries of industrial business process models.

6.4. The choice of the techniques and their priority

The choice and prioritization of the techniques used to build a control-flow analyzer should be tailored to the type of business process models analyzed. Some languages like BPEL (39) have a block oriented structure and encourage the use of well-formed cycles only. The well-formed cycles, as opposed to arbitrary cycles, can always be factored out using a decomposition technique and their correctness can be checked using simple heuristics. Therefore symbolic execution combined with heuristics is appropriate to analyze BPEL models. Most modeling languages support 'spaghetti-like' business process models where arbitrary cycles are commons. If the language does not contain IOR-joins or other control-flow constructs that cannot be mapped easily to a free-choice workflow net, then structural analysis alone would be perfectly suited. For business process models containing arbitrary cycles and IOR-joins a more complete control-flow analyzer, such as the one presented in Sect. 6.3, is required.

The arrangement of techniques presented in Sect. 6.3 is designed to cover all business process models whose control-flow can be modeled by workflow graphs, which may contain IOR-joins. The techniques are prioritized to optimize the consumability of the diagnostic information and the efficiency of the analysis, i.e., when possible, we use the structural analysis and symbolic execution which provide better diagnostic information and have higher efficiency than state space exploration.

We prioritize consumability over efficiency. For example, we refrain to use negative heuristics, which are extremely efficient, to keep a more homogenous diagnostic information: All techniques that we use provide an error trace. In addition to a reduced error trace, the symbolic execution and structural analysis provide additional diagnostic information. We will describe the diagnostic information in more details in the following section.

We make a pretty arbitrary choice in prioritizing the structural analysis over symbolic execution. As both techniques provide complementary diagnostic information, they could also be combined for the analysis of many fragments, i.e., we can check acyclic fragment without IOR-join using both techniques and offer a combined diagnostic information. We do not systematically use both techniques on acyclic fragments without IOR-join, but envision to allow the modeler trigger a complementary symbolic execution to obtain the additional diagnostic information.

We will evaluate in Chap. 7 whereas the chosen combination of techniques satisfies our efficiency, consumability, and coverage requirements on a large set of industrial business process models.

6.5. The error display and fix suggestion

In this section, we first present a mock-up user interface which allows us to describe the envisioned user interaction. Then, we summarize the different types of error information that we obtain from our analyzer. We present suggestions to fix an error that we can derive directly from the error display and finally discuss how support the dismissal of some errors.

6.5.1. Tool mock up

We present a mock up of a control-flow analysis integrated in an editor for business process models. We proceed with a mock-up rather than with print-screens of specific tool because it gives us more freedom to present our complete vision without the technical or graphical limitations of a particular tool[1]. Figure 6.7, shows a view of a tool (35) allowing to model a business process. The control-flow analyzer is run during the modeling: at every new connection (or connection removal). The location of a control-flow errors is indicated on the canvas by a red cross. Because the control-flow errors appear when they are created, i.e., as soon as a connection creates a control-flow error, the user receives immediate feedback which is already useful to understand the error: the last connection created the error.

Upon right clicking on an error, a contextual menu provides some further operation to deal with the error. Figure 6.8 illustrates the result of selecting the color trace option of the contextual menu of one of the errors. In this example, it is a simple deadlock resulting from the incorrect pairing of an XOR-split and an AND-join.

The deadlock of Fig. 6.8 occurs in every execution executing this portion of the process, i.e., for every customer that is not a gold customer. As described in Sect. 5.3, our analysis detects that this error always occurs and therefore does not allow this error to be dismissed by the user. One fix is, for example, to replace the final AND-join by an XOR-join.

[1]While we implemented numerous different analyzers tailored to different modeling languages, none of the implementations uses the full set of techniques, types of diagnostic information, and fixing support that we envision. Multiple reasons contribute to this: Depending on the problem and modeling language, not all the techniques are necessary. The structural analysis technique presented in Chap. 3 is new and there is only one implementation of it in a research prototype. Some types of diagnostic information do not adhere to the design directions of some our tools. The support to fix control-flow is implemented only in a prototype (7) independent of this work. We will discuss the different tools created and the techniques and diagnostic information that they implement in details in Sect. 6.6. We will present some real output of one of our tool in Chap. 7.

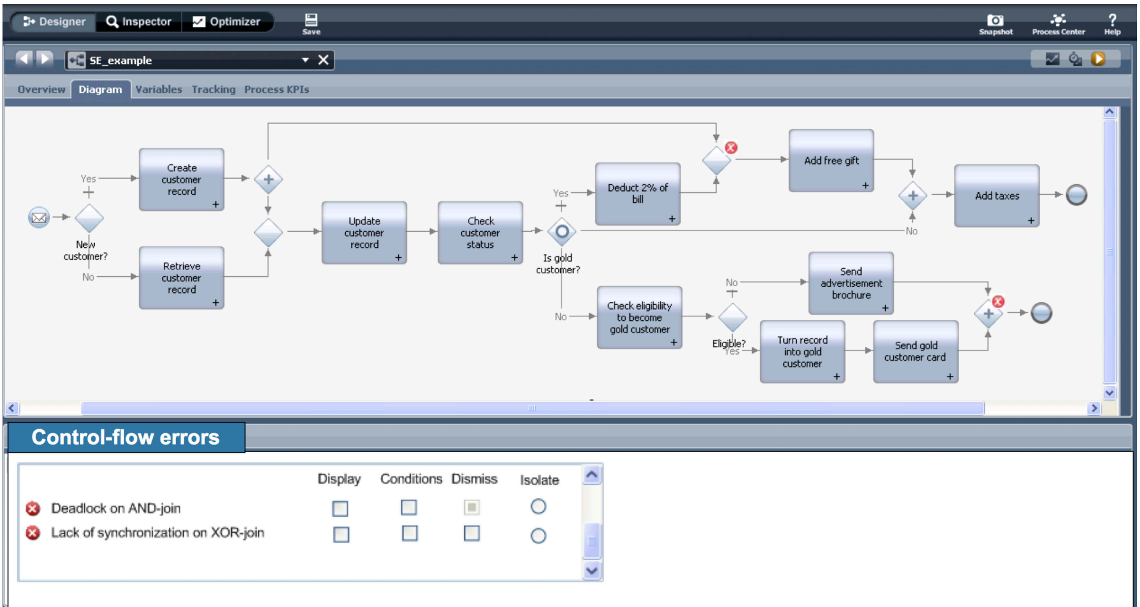


Figure 6.7.: Errors appear on the canvas during the modeling phase.

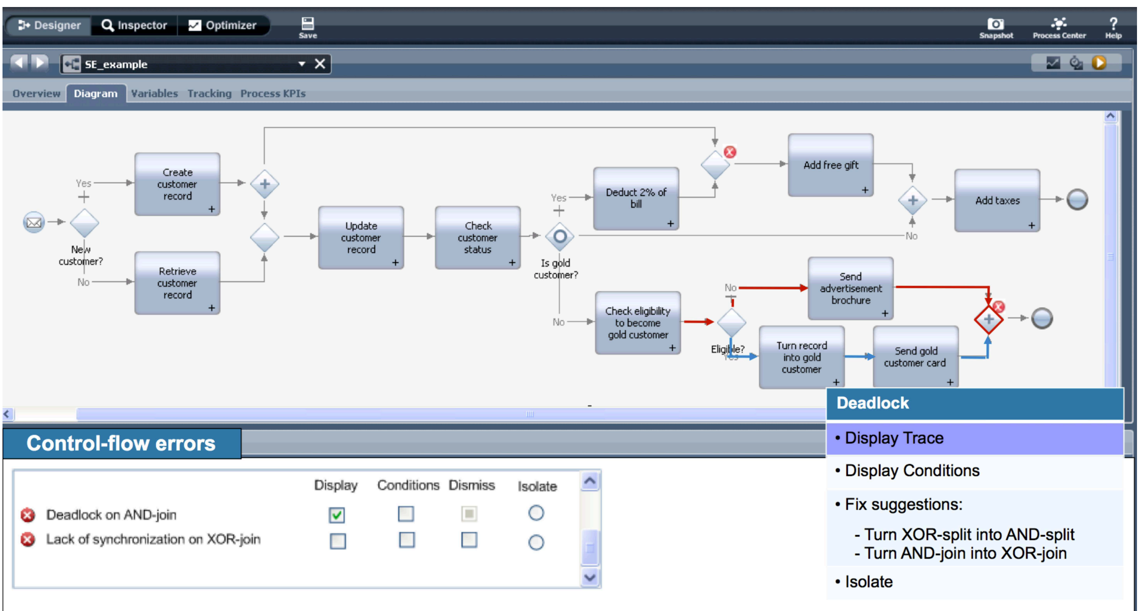


Figure 6.8.: Displaying an error trace for the deadlock.

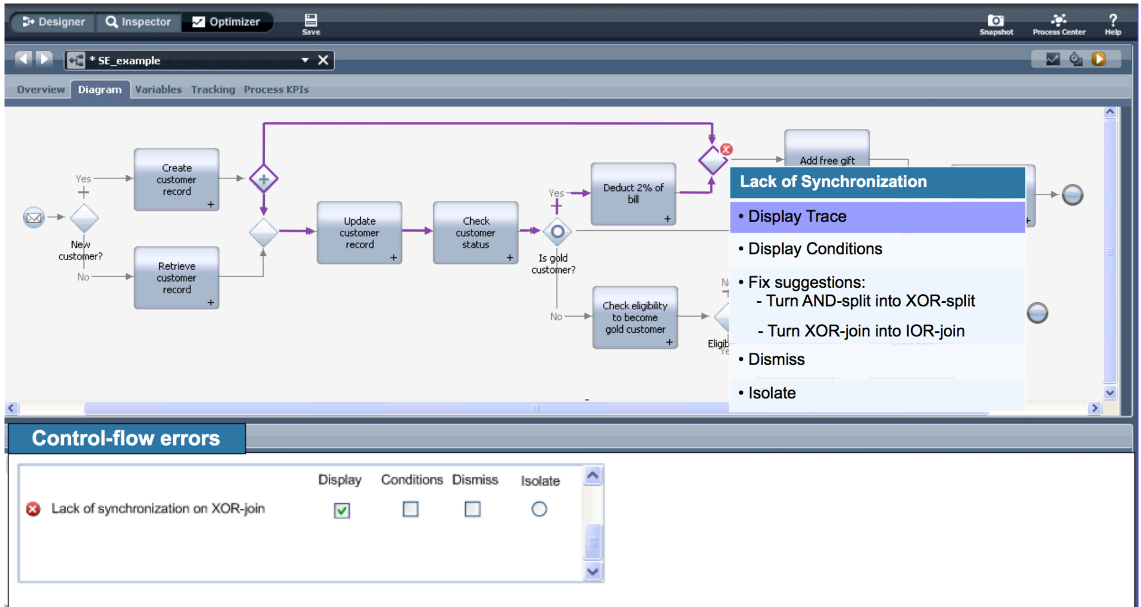


Figure 6.9.: Displaying an error trace for the lack of synchronization.

Figure 6.9 shows the error trace displayed for the second error in the process. This error is a lack of synchronization.

Unlike the first error, this second error is not trivial to understand. To understand the error, the user can choose to display the execution conditions under which the error occurs. Figure 6.10 shows the result of displaying these error conditions. The user can observe from the displayed error conditions that the first incoming edge of the XOR-join where the lack of synchronization is located carries a token every time the condition “is new customer” is evaluated to true during the execution whereas the second incoming edge carries a token in every execution where “is gold customer” is evaluated to true. The user, who has knowledge about the domain, can identify that, in his context, a new customer cannot be a gold customer already. Therefore, this lack of synchronization never occurs in an actual run of the process and can be dismissed. Note that the contextual information allowing the user to dismiss this error is not explicitly modeled. Therefore any control-flow analyzer will report such control-flow error.

After the dismissal of this error, the analysis continues and possibly finds further errors in this process.

This example process was small enough for the errors to be easily localized and understood. Larger examples are common in practice (See Chap. 7 for the typical size of an industrial business model). Fragments can be leveraged to isolate the error by hiding fragments that do not contribute to the error as described by Vanhatalo in his PhD thesis (72).

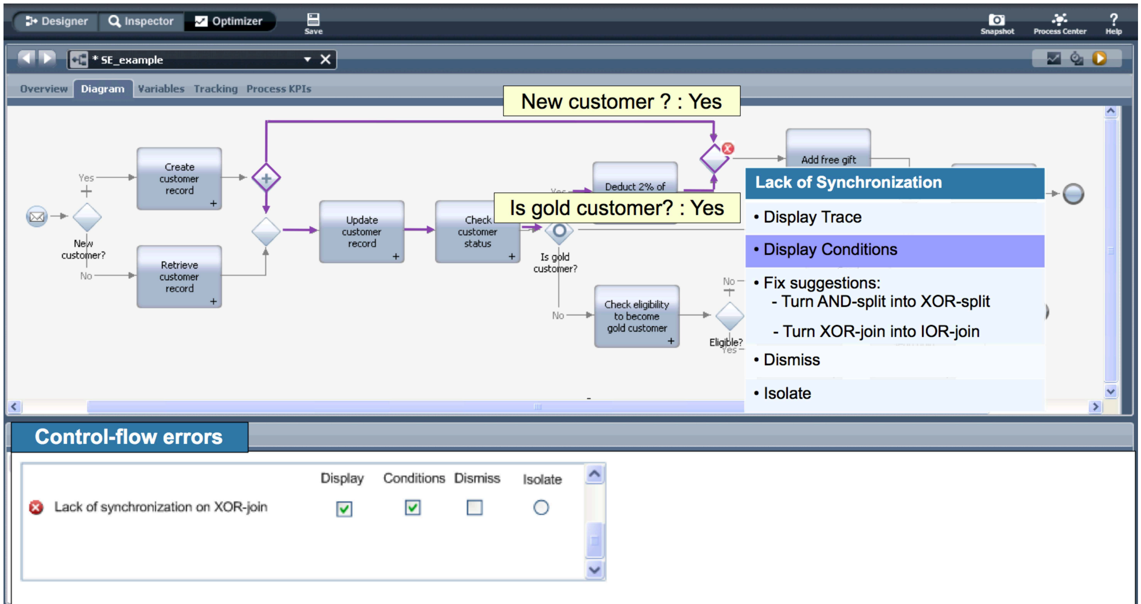


Figure 6.10.: Displaying the conditions leading a trace into a lack of synchronization.

6.5.2. Error display

Control-flow errors can stem from various causes such as slips during the modeling, misunderstanding of the semantics of the modeling language, incomplete understanding of the process being modeled or the process simply becoming too complex to keep track of its control-flow. The first step toward fixing a control-flow error is to locate and understand it. We now review the error displays available for each analysis path of Fig. 6.6, i.e., we summarize the diagnostic information returned by each of the three analysis techniques that we use:

Structural analysis: The structural analysis, described in Chap. 3, returns patterns that can be highlighted on the business process model as diagnostic information. Moreover, we described how each structural pattern can be used to obtain an error trace in polynomial time.

Symbolic execution and acyclic handle check: The symbolic execution, described in Chap. 5, provides execution conditions that characterize the execution(s) that lead to a control-flow error. Moreover, the conditions allow us to derive an error trace. In case of a lack of synchronization, the acyclic handle can be, similarly to the patterns identified by the structural analysis, displayed as diagnostic information and used to derive an error trace.

State space exploration: The state space exploration, described in Sect. 6.2.4 returns an error trace as diagnostic information.

The different analysis techniques potentially provide to the user different diagnostic information. In order to harmonize the diagnostic information, we chose to center the diagnostic information around the error trace that can be derived from the result of all of our analysis techniques.

Note that, as argued in Chap. 1, the error traces returned by state space exploration might contain significantly more steps than necessary. We partially reduce these error traces by analyzing fragments in isolation using the RPST: the error traces do not contained transitions executing nodes located outside of the fragment and therefore are not relevant to the error. The error traces derived from structural patterns or error conditions are potentially even more concise because, as suggested by the proofs in Sect. 3.3, we use the error pattern to direct the construction of the error trace and thus only construct the relevant portion of the trace. Moreover, the portion of the error trace to reach the edge preceding the P/T-handle of the siphon, the T/P-handle of a path to the final place, or the start of the acyclic handle can also be omitted in order to reduce the error trace further. While we did not claim or prove any minimality condition on the error traces returned by our analyzer, we still call them *reduced error trace* due to their potentially smaller size than error traces returned by plain state space exploration.

As described in Sect. 6.2.2 and justified in Sect. 6.4, we chose to not use negative heuristics to detect control-flow errors in this analyzer. However, they provide a complementary means to understand some control-flow errors as a violation of modeling rule. For example the error message ‘The highlighted fragment contains a cycle but only contains AND logic gateways, therefore it must contain a deadlock’ is an error message displayed when a particular heuristics is violated. This additional means to understand a control-flow error can be displayed as an additional diagnostic information. This type of diagnostic information has been used in former versions of our analyzer. For large business process models, this diagnostic information is helped by the isolation capability provided to us by the fragment decomposition.

To summarize, for any control-flow error, our analyzer provides an error trace. More detailed information is available under certain conditions, for example to get the error conditions provided by symbolic execution, the process must be acyclic. We will see in Chap. 7 the frequency under which a fragment of an industrial business model can be analyzed by any of our techniques and what is the proportion of error detected by each technique. This will allow us to derive the frequency under which each diagnostic information can be obtained.

6.5.3. Fix suggestions

Once an error is understood, fixing it might require as little work as changing the logic of a gateway. It might also require a major refactoring of the control-flow of the process. In the following, we describe fix suggestions that can be proposed to a user in order to fix a control-flow error. These suggestions are directly derived from the diagnostic

information of our analysis techniques.

We have observed in a study (19) that many errors occur in fragments that contain only two gateways. These gateways are wrongly paired such as, in the case of the error highlighted in Fig. 6.8, where the XOR-split is wrongly paired with an AND-join. In this case, the wrong pairing of the gateways results in a deadlock. Although wrongly paired gateways seem rather easy to spot, they cause roughly 30% of the control-flow errors detected (19). While we chose to not use negative heuristics to detect control-flow errors (cf. Sect. 6.2.2 and Sect. 6.4), simple heuristics can identify wrongly paired gateways and provide a simple fix suggestion: one can change the logic of either the splitting gateway or the joining gateway to accommodate the logic of the other. Fix suggestions for the lack of synchronizations identified by an acyclic handle (Sect. 5.2) can be derived using the same logic.

The deadlocks identified by the symbolic execution technique (Chap. 5) lead to three types of fix suggestions. The first can be proposed if the two edges which do not carry an equivalent symbol, are mutually exclusive. This can be checked easily using the check for acyclic handles (Sect. 5.2). If the edges are not mutually exclusive, the deadlock can be fixed by merging these edges using an XOR-join. A second fix suggestion can be derived from the symbols assigned to the edges: we can bridge (in the sense of the bridges described for the K-replacement (Sect. 4.5)) the necessary decision outcomes to the respective edges in order to ensure that they carry an equivalent symbol. The third fix is simply to replace the AND-join that is the target of the edges, i.e., the AND-join where the deadlock is detected, by an IOR-join.

The patterns identified by the structural analysis of Chap. 3 also provide candidates for fixing the control-flow of a process. In case of a simple path to the final place with a T/P-handle (in the context of a workflow graph, an AND/XOR-handle) is identified, the logic of the first or the last gateway of the handle can be changed, i.e., the AND-split turned into an XOR-split or the XOR-join turned into an AND-join or eventually an IOR-join. In case of a DQ-siphon with a P/T-handle (XOR/AND-handle) we can also change the logic of the XOR-split or the AND-join. Moreover, adding AND/XOR-bridge from the handle to the siphon might fix the error.

While applying a fix suggestion derived from wrongly paired gateways ensures to fix the error, applying some of the other fix suggestions might introduce a new error. For example, consider the handle highlighted in Fig. 6.9. Replacing the XOR-join at the end of this acyclic handle by an AND-join would introduce a deadlock occurring at the newly inserted AND-join. It is desirable to prioritize the fix suggestions that do not introduce a new error such as, for example, turning the XOR-join into an IOR-join.

One can go further in prioritizing the fix suggestions by ranking them. In his master thesis (7), Deibler developed a ranking metric for fix suggestions derived from acyclic T/P-handles, the result state space exploration, and some of the heuristics. The proposed metric ranks first the fix suggestions that modify the logic of the smallest number of gateways. The metric also gives a better rank for fix suggestions that do not introduce an IOR-gateway and favors fix suggestions modifying joins over those modifying

splits. This approach can be easily extended to rank all the fix suggestions produced by our control-flow analyzer.

In the course of this research, we observed that some frequent errors can be caused by a particularity of an editor or even a modeling language concept that is often misunderstood by the user. For example, we observed during a study (19) that an error was occurring frequently due to a misunderstanding of the instantiation semantics when using multiple start events. In such cases, some fix suggestions might be tailored to address the common errors occurring in a particular editor and the ranking could be adapted and promote the appropriate fix suggestions.

The goal of the fix suggestions mentioned in this section is to provide a direct fix for as many errors encountered in practice as possible. To evaluate the quality of our fix suggestions, a study involving practitioners would be required to assess at which frequency the fix suggestions actually provide a desirable fix, i.e., a fix that a practitioner would actually apply to fix the control-flow error in contrast to a fix that results in a sound process. Such study is out of scope of this thesis and is left as future work.

6.5.4. A note on error dismissal

As described in details in Sect. 5.3, some errors are detected but may not arise in a real execution of the process due to the correlation of data-based decisions. This is the case for any control-flow analyzer.

We described in Sect. 5.3 how symbolic execution allows the user to reason about the errors in order to determine the ones that do not occur in any real execution, dismiss them and continue the analysis. Such support is unique to symbolic execution and can therefore only be provided for errors occurring in acyclic fragments. For errors occurring in cyclic fragments, we allow the user to dismiss an error but there is no further analysis for this fragment.

6.6. Tools developed

During the course of this work numerous control-flow analyzer prototypes have been implemented for various modeling tools. In particular, the architecture presented in Sect. 6.3 has been implemented as a research prototype in IBM WebSphere Business Modeler (36). This latter prototype includes the techniques of Chap. 3 and Chap. 5 and their corresponding diagnostic information. A separate research prototype implementing some of the fix suggestions described in Sect. 6.5.3 was implemented by Thomas Deibler in the context of his master thesis (7).

In the following, we present the versions that have been consolidated into industrial tools and deployed in IBM products.

6.6.1. Plugins for IBM WebSphere Business Modeler

The first implementation of the control-flow analysis to be released was based on state space exploration. The control-flow analyzer plugin was included in a set of plugins for the IBM WebSphere Business Modeler (36) called *Accelerators*. These plugins were released on IBM developerWorks (34) together with a series of articles. The goal of the accelerators is to support the user in creating business process models of higher quality faster. The plugins were also demonstrated (24) at the Business Process Management (BPM) conference of 2009.

This implementation uses the refined process structure tree and the heuristics to speed up the analysis. The analysis can be triggered by the user at any time during the modeling, on a chosen set of processes. The errors identified in the analyzed process(es) are listed and, for each error, a textual error message describing the error is provided. The plugin also allows the user to highlight the fragment containing a particular error in order to localize the error more easily in large processes.

6.6.2. Validation in IBM WebSphere Business Modeler

Starting with version 6.2, IBM WebSphere Business Modeler (36) includes our control-flow analyzer. It has two main use cases: It is a part of the validation occurring before translating a business process model described using the IBM WebSphere Business Modeler modeling language into a BPEL (39) model. This translation allows the user to export their model to IBM WebSphere Integration Developer (37). The same control-flow analyzer is used as continuous validation by the so-called direct-to-deploy feature, which allows a user to directly deploy a IBM WebSphere Business Modeler model to IBM WebSphere Process Server (38). I.e., in a particular mode, the analysis is triggered automatically every time the process is saved.

This control-flow analyzer was initially based on the code base of the accelerators plugin described in the previous section. It was consolidated over time and now also features the symbolic execution and acyclic handle check described in Chap. 5. The control-flow errors are listed and described textually, together with other validation errors, in an error view.

6.6.3. Validation in IBM WebSphere Integration Developer and IBM WebSphere Process Server

IBM WebSphere Integration Developer(37) also uses a version of our control-flow analyzer to validate BPEL processes before deploying them on IBM WebSphere Process Server(38).

The analyzer code is the same as the one used in IBM WebSphere Business Modeler. The only parts that differ from the IBM WebSphere Business Modeler implementation are the component translating the control-flow of the process (in this case a BPEL

process) into a workflow graph and the component in charge of the error display. These two components are, in general, the only components which are editor specific.

Over time, the code migrated to become a component of IBM WebSphereProcess Server. This component is still invoked by IBM WebSphere Integration Developer for its validation as described above. In addition it is also used directly by the process server as “last line of defense”, i.e., to provide a last warning to a user who wants to deploy a process containing a control-flow error. In case an error is identified, the error message is a text message printed in the server console.

6.6.4. A note on the inclusion of structural analysis

The structural analysis (Chap. 3) is only implemented in our research prototype. This technique was the latest developed and did not reach a release plan yet. The control-flow analyzer remains a candidate feature for other products. We hope to see structural analysis included in the future releases of the existing tools and other products.

6.7. Related work

In this section, we discuss the related work on control-flow analyzers, especially control-flow analyzers based on state space exploration.

The pioneering work in control-flow analysis (66, 67) resulted in the business process verification tool Woflan (79). Woflan uses a combination of Petri net analysis techniques, most notably structural Petri net reduction (53) and S-coverability analysis, and a form of state space exploration, to check whether a workflow net, which can be non-free-choice, is live and bounded. Note that this problem is different from checking soundness of free-choice worklow nets, which we solve in Chap. 3. As Woflan uses a combination of techniques, the diagnostic information it returns depends on the technique that detected the control-flow error. Verbeek et al. (79) report that, besides detecting mismatches (cf. discussion in Sect. 3.7), error traces provide the most useful diagnostic information for correcting an error.

LoLA (82) is another tool based on state space exploration used to decide numerous properties of a given Petri net by an inspection of its state space. Among other techniques, LoLA uses partial order reduction techniques (44) to mitigate the state space explosion problem. LoLa is integrated in several Petri net verification platforms (32, 58, 69). LoLA has also been applied to other control-flow analysis problems, for example, it is used to analyze web service collaborations (48). Upon detecting a state violating the property being checked, LoLA produces an error trace leading to this violating state.

In a case study (18, 19), we compared Woflan, LoLA, and the control-flow analyzer developed during the master thesis (20) of the author of this thesis. The later analyzer combined process structure tree decomposition (cf. Sect. 6.2.1), heuristics (cf. Sect. 6.2.2), state space exploration (cf. Sect. 6.2.4). The main question addressed in

this study was to check if the soundness of industrial business process model could be checked fast, i.e., in less than a second. The study answered this question affirmatively. The consumability of the error display returned by the tree tools, which was not the main focus of study, turned out to be less satisfactory, which motivated our work on structural analysis and symbolic execution.

State space exploration, usually combined with reduction approaches (53), is often used to support various control-flow analysis use cases and studies (52, 83). For example, Mendling et al. (52) use state space exploration to check relaxed soundness of Event Process Chains (EPC). While not focused on analyzing business process models during the modeling phase and returning the diagnostic information to the modeler, such analyzers can provide an error trace as diagnostic information.

7

Evaluation

In Sect. 6.3, we described how we combined the techniques presented in this dissertation to obtain a control-flow analyzer.

In this chapter, we use this analyzer to perform a control-flow analysis of more than 1350 industrial business process models and evaluate whether the analyzer satisfies the requirements formulated in Sect. 1.2.2:

1. *Efficiency:* Do we obtain the results fast enough? We check whether, in practice, our analyzer performs the analysis of an industrial business process model in less than a second.

2. *Consumability:* Can we provide diagnostic information? More precisely, we check at which frequency can we provide our best diagnostic information: a structural error-pattern combined with a reduced error trace. How often do we get the diagnostic information provided by symbolic execution? How often do we have to resort to an error trace returned by state space exploration?

3. *Coverage:* Can all business process models be analyzed?

As discussed in Sect. 6.4, while we described and implemented an comprehensive analyzer, the choice of the techniques should be tailored to the features of the modeling language to be analyzed. We will therefore also discuss the most appropriate set of techniques for the models analyzed in this chapter.

In Sect. 7.1, we introduce the data that we chose for the experiments. In Sect. 7.2, we describe the experimental setup. In Sect. 7.3, we present and analyze the results of the experiments. In Sect. 7.4, we summarize our results.

7.1. The data set

We reuse a subset of the data set that we used in two previous publications (18, 19). We chose this data set because it regroups a large number of 'real-world' business process models, i.e., process modeled by professional in the course of industry projects.

In the course of our previous work, we considered a set of more that 3000 business process models coming from IBM global services, which were captured with the goal to implement them in a service-oriented architecture.

The models cover various industry domains such as financial services, automotive, telecommunications, construction, supply chain, health care, and customer relationship management. We also looked at large collections of reference processes that were created using different modeling styles.

All the business process models were available in the IBM WebSphere Business Modeler (36). They are modeled in a language that combines elements from UML Activity Diagrams (17) and the Business Process Modeling Notation (BPMN) (54). Some of the models were created with other tools first and then imported into the IBM WebSphere Business Modeler.

The modeling language does not contain IOR-joins per se. However, it does support multiple end nodes which, in order to be translated into a workflow graph, need to be merged. This merging can be achieved using an IOR-join. This final IOR-join can, as we described in Sect. 6.2.3, always be replaced using only XOR and AND gateways.

Like in our previous experimental studies, we reduce our initial test set from approximately 3000 models to 1350 models. The reason for this reduction is that only some of the model collections are useful to test our analyzer: As good modeling practices suggest, large processes are usually decomposed into subprocesses. This decomposition often results in many small business process models that are free of control-flow errors. These business process models are neither interesting nor challenging for this evaluation. Some other business process models, in particular those created using other tools and then translated to be used in IBM WebSphere Business Modeler, were not created with the appropriate notion of soundness in mind or had been created by novice users and, as a consequence, were syntactically incomplete. These process models are flawed in such a way that it does not make sense to consider them for a control-flow analysis.

The 1350 models selected for this evaluation are distributed in 4 libraries called A, B1, B2, and B3. The library A contains process models from the insurance domain. Libraries B1, B2, and B3 contain a series of models from the banking domain. Table 7.1 characterizes the size of the processes in the libraries in terms of the number of nodes and edges that they contain.

The models of the B series partially overlap: They were created over a period of two years, in which a library changed to the next by adding more process models and refining all models. Thus, out of the 1350 industrial processes models that we analyze,

	A	**B1**	**B2**	**B3**
Avg. / max. number of nodes	14 / 46	17 / 69	16 / 67	18 / 83
Avg. / max. number of edges	33 / 127	29 / 147	31 / 202	33 / 195

Table 7.1.: Static data of the process libraries in the case study.

it can be considered that 700 models[1] represent completely unique business process models. The remaining 650 models contain earlier versions of the models which vary from their later version in modeling style and level of details.

7.2. Setup

The analyzer presented in Sect. 6.3 is implemented in Java. To conduct the experiments it is packaged as a plugin of IBM WebSphere Business Modeler.

The experiments are conducted on a notebook with a 2 GHz processor and 2 GB RAM. The analysis times are computed as an average over 10 runs. It also includes the time spent by the tool to generate the error report for the user. The overhead for loading the process models from the hard drive into memory during the first run is factored out from the analysis time.

We validated the correctness of the results, i.e. the detected soundness or unsoundness of the processes, by comparing them with the results obtained during previous experiments (18, 19) which were conducted on the same set of processes using different analysis techniques.

7.3. Analysis

Table 7.2 presents the results obtained when analyzing our test libraries using the analyzer described in Sect. 6.3. In the following, we discuss these result according to the efficiency, consumability, and coverage criteria.

7.3.1. Efficiency

As shown in Table 7.2, it takes approximately a second for the analyzer to run on any of the libraries, i.e., to perform the analysis of approximately 400 industrial business models. The analysis of a single business process model takes only a few milliseconds and we observed that no fragment required more than 32 milliseconds to be analyzed. The speed of the analysis allow us to run a control-flow analysis frequently during the modeling of a business process model: we could even perform a control-flow analysis of

[1] This number is obtained by summing the number of the processes in the latest library of the B series, which is B3 with 421 processes, and the 282 processes in library A.

	A	B1	B2	B3
Number of processes (Sound-Unsound)	282 (152-130)	287 (107-180)	363 (161-202)	421 (207-214)
Sound fragments	10'514	11'097	13'870	17'250
Unsound frag. identified by struct. analysis	134	252	301	301
initially empty siphon	0	17	15	13
path with a T/P handle	50	185	223	246
siphon with a P/T handle	84	50	63	42
Resort to SE and HC	0	0	0	0
Resort to SSE	0	0	0	0
Average library analysis time [ms]	823	856	968	1194

Table 7.2.: Experimental results for the analyzer described in Sect. 6.3 using, in particular, structural analysis (cf. Chap. 3), symbolic execution (SE) combined with handle checking (HC) (cf. Chap. 5), and state space exploration (SSE) (cf. Sect. 6.2.4).

	A	B1	B2	B3
Number of processes (Sound)	282 (152)	287 (107)	363 (161)	421 (207)
Unsound processes identified by struct. analysis	130	180	202	214
initially empty siphon	0	17	15	13
path with a T/P handle	47	138	158	174
siphon with a P/T handle	83	25	29	27
Average library analysis time [ms]	752	492	627	928

Table 7.3.: Experimental results for the analyzer using structural analysis and completion only.

the business process model on every editing step without the analysis being noticed by the user.

The analyzer solely uses the structural analysis. Given that we can model the control-flow of these models without IOR-joins, this is not surprising. It shows the suitability of the structural analysis combined with the Kiepuszewski completion (cf. Sect. 6.2.3) to analyze this data set. In fact, a simplified version of this analyzer, based only on these two techniques, would be sufficient to analyze the business processes modeled with the IBM WebSphere Business Modeler.

We run a second set of experiments using only the structural analysis and the Kiepuszewski completion, i.e., without using the process structure tree decomposition, the heuristics, the symbolic execution, and the state space exploration. Table 7.3 shows the results of this experiment.

We observe that, in this simplified configuration, the analysis time per library goes down. The difference in time is due to the removal of the computation of the process structure tree. It is more efficient to analyze the whole process at once with the struc-

tural analysis than to compute the process structure tree and analyze each fragment in isolation. However, as we cannot analyze each fragment in isolation anymore, we detect less control-flow errors. We still detect the same number of unsound processes but, as the analysis stops upon detecting the first error, we detect at most one error per process whereas the decomposition allowed us to find multiple independent errors per process.

In earlier case studies (18, 19), we compared three different control-flow analyzers, which use state space exploration in combination with different optimization techniques to counter the state space explosion problem. In comparison with these analyzers, our control-flow analyzer is roughly 2 to 6 times faster. Note that the speed of these other analyzers was already sufficient to perform a control-flow analysis while modeling, and, while the speedup provided by the structural analysis is significant, it is not game changing. However, as already mentioned, we also recognized in these earlier studies that the consumability of the diagnostic information returned by these other analyzers needs improvement (cf. Sect. 6.7).

In this section, we have shown that the analyzer we presented in Chap. 6 is efficient enough to be used, without being noticed, while the modeler is editing the process. The efficiency was the first requirement. In the next section, we will check the two other requirements: coverage and consumability.

7.3.2. Coverage and consumability

In Sect. 6.3, we designed an analyzer capable of analyzing any workflow graph. As we can translate the control-flow of every process of the data set into a workflow graph, our analyzer achieves full coverage of the process models in the data set. The coverage of this analyzer extends to any process modeled with the IBM WebSphere Business Modeler.

The analyzer can analyze any workflow graph. However, it does not always provide the same type of diagnostic information: In the best cases, the analyzer provides a structural error-pattern and a reduced error trace. In case of an error in a cyclic workflow graph fragment containing an IOR-join, the analyzer resorts to state-space exploration and thus only provides an error trace as diagnostic information. In the following we check at which frequency our analyzer can provide each type of diagnostic information.

As observed already in the previous section, all the process models of this data set can be analyzed using structural analysis due to the lack of IOR-joins in the model. Therefore, the analyzer always provides a structural error pattern and a reduced error trace (cf. Sect. 6.5), i.e., our best case diagnostic information. In the following, we first show a few examples of the structural error patterns detected. Then, we will change a bit the experimental setup to gain more insights on the frequency at which the analyzer provides each type of diagnostic information when the model contains IOR-joins.

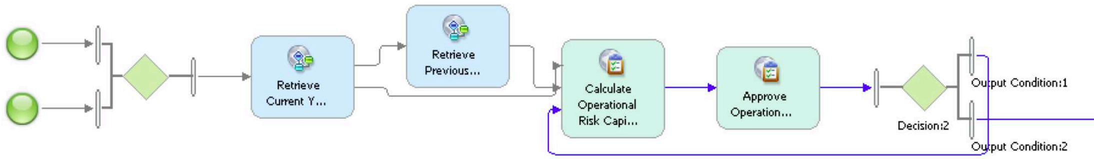


Figure 7.1.: Screenshot of the display of a siphon that is not initially marked. The siphon (in blue) does not contain a start event. Thus, by the siphon property (Proposition 3.3), the siphon is never marked, i.e., the portion of the model colored in blue cannot be executed.

Fig. 7.1 shows the diagnostic information[1] provided by the control-flow analyzer to report a siphon that is not initially marked, which is colored in blue. As discussed in Chap. 3, this siphon cannot get marked during any execution of this process.

In this example, as well as the examples that follow, we see that activities can have multiple incoming and outgoing edges in the modeling language. The control-flow logic between these multiple incoming and incoming edges can be specified in the properties of these activities. By default, the logic between the edges is the AND logic, i.e., multiple incoming edges of a tasks have the same semantics as if they were joined by an AND-join which was then connected to the task and multiple outgoing edges have the same semantics have if they were coming out of an AND-split. For simplicity, we select example of process models using the default (AND) logic for the remainder of this chapter.

Fig. 7.2 shows the diagnostic information provided by the control-flow analyzer to report a simple path to the final place (colored in blue) with a T/P handle (colored in red). Fig. 7.3 shows a DQ-siphon (colored in blue) with a P/T handle (colored in red).

To accommodate the limited resolution available and the absence of scroll bar on a sheet of paper, as opposed to a computer screen on which these models are usually visualized, we present only portions of the business process models in which the errors occur and only select errors that are 'compact', i.e., for which the error pattern is small. These examples might mislead the reader into thinking that these errors are easy to spot. However, even these compact errors are easily overlooked: As the author experienced while writing this chapter, even knowing that a given process contains a given type of error pattern, it does take time to locate the error pattern if one does not use the control-flow analyzer to color it.

In cases of larger error patterns displayed in large and complex business process mod-

[1]The prototype also provides a view allowing the user to navigates through the process analyzed, notice the process containing an error, obtain a textual description of the error in the process, and ultimately to display the diagnostic information related to a control-flow error on a business process model itself, which is what we display in the illustrations of this chapter. We do not discuss the textual diagnostic description and the navigation through the errors in the process further.

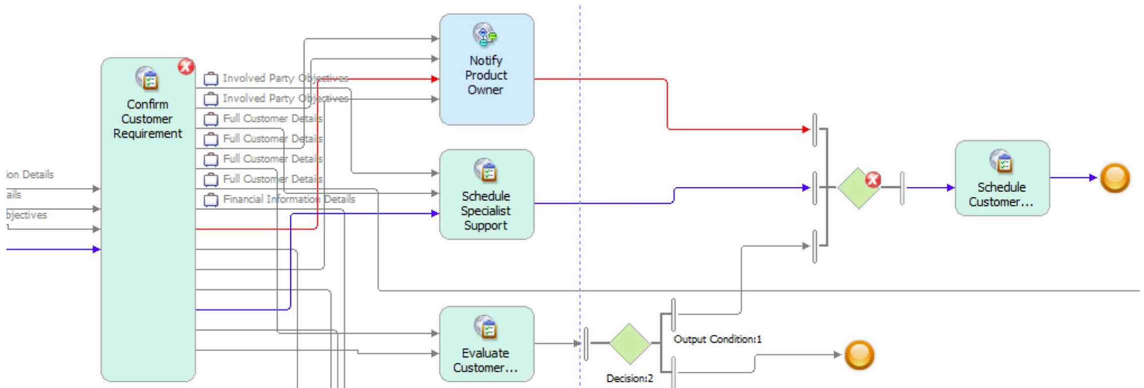


Figure 7.2.: Screenshot of the display of a simple path to the final place with a T/P handle. The activity 'Confirm Customer Requirement' is on a simple path (blue) to the final place and starts multiple concurrent paths including the T/P handle (in red). Both concurrent paths join on the XOR-join (empty diamond with error flag) without being properly synchronized.

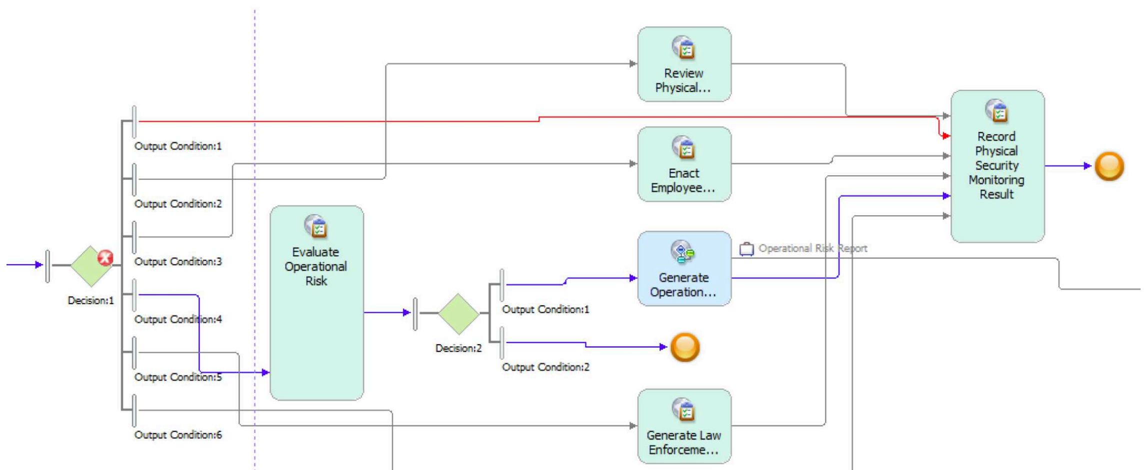


Figure 7.3.: Screenshot of the display of a DQ-siphon with a P/T handle. Once the first outcome of 'Decision:1' is chosen, i.e., the first transition of the P/T handle (in red) is executed, the DQ-siphon (in blue) is emptied of its only token. The token on the handle is stuck because the activity 'Record Physical Security Monitoring Result', which is pat of the empty DQ-siphon, cannot be executed.

	A	B1	B2	B3
Number of processes (Sound-Unsound)	282 (152-130)	287 (107-180)	363 (161-202)	421 (207-214)
Sound fragments	8941	9311	11744	15050
Fragments completed with IOR	282	288	311	355
Errors fragments identified by struct. analysis	6	142	172	169
Errors fragments identified by SE and HC	97	104	124	124
Error fragments identified by SSE	31	13	11	12
Average library analysis time [ms]	716	832	1211	1384

Table 7.4.: Experimental results using an IOR-join to complete business process models with multiple end nodes.

els, we observe that the error patterns retain the same level of consumability because, to understand the error, the modeler only needs to visualize the structure of the error pattern, as opposed to multiple elements that compose it, and the two intersection points of the handle. In comparison, we find that error traces become more difficult, or at least more time-consuming, to analyze when they become larger. Moreover, the size of the error trace tends to increase when the size of the business process model increases. Therefore, we believe that structural error patterns have a scalability advantage over the error traces.

While the analyzer only performs a structural analysis on this data set, it is also possible to perform an additional symbolic execution on the fragments containing an error to obtain a complementary diagnostic information. A symbolic execution can be performed an any acyclic fragment. More than 90% of the fragments containing an error in the libraries of the B series and on more than 70% of the fragments containing an error in the library A are acyclic and, therefore, can be analyzed with the symbolic execution.

In order to get more insights on the frequency of use of each technique in business process model containing IOR-joins, we now perform an additional set of experiments during which we do not replace the final IOR-join used to merge multiple end events. Table 7.4 summarizes the results of this experiment.

In this setup, the other symbolic execution and state space exploration become necessary to perform the analysis. This indicates their potential value for the analysis of business process models containing IOR-joins. For the libraries of the B series, the distribution of the techniques used to identify and display the erroneous fragments is roughly 55% structural analysis, 40% symbolic execution, and 5% state space exploration. For the analysis of the A library, this distribution changes to 4% structural analysis, 72% symbolic execution, and 23% state space exploration in the A library.

The distributions between the techniques used differs significantly between the library A and the libraries of the B series. In the library A, only a few errors are identified

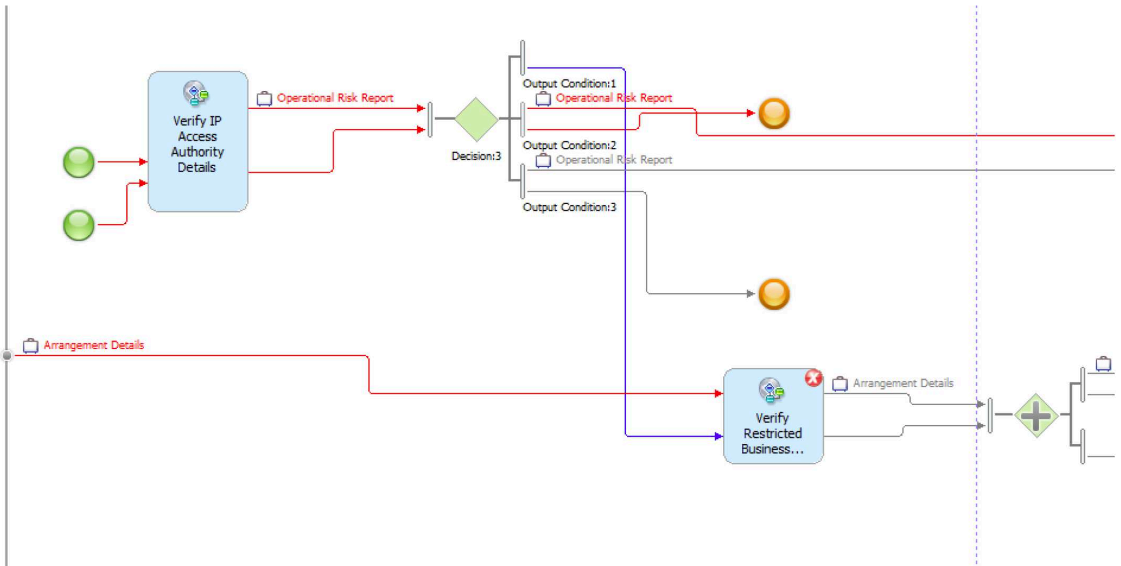


Figure 7.4.: Screenshot of a portion of a process modeled using multiple end events to stop various execution threads. This modeling style results in a large fragment, containing an IOR-join, which needs to be analyzed using symbolic execution or state space exploration. In this case, the process contains a deadlock whose error trace, obtained using symbolic execution, is displayed in red.

using structural analysis. A reason for the low applicability of structural analysis in library A is the combination of a particular modeling style used with the way merge multiple end events with an IOR-join. Fig. 7.4 illustrates a business process modeled using this particular modeling style. This modeling style uses multiple end events at various points of the process to stop the execution, or at least an execution thread. In order to translate such a business process into a workflow graph with a single sink, the multiple end events are connected to a final IOR-join. These additional connections from multiple points of the process model result in a large fragment containing an IOR-join. This last fragment is analyzed using symbolic execution if it is acyclic, state space exploration otherwise. These large fragments also appears to contain most of the errors. The business process model illustrated by Fig. 7.4 contains a deadlock detected by symbolic execution.

To summarize this section, the control-flow analyzer presented in Chap. 6 is able to cover all the business process models in the data set and provide the diagnostic information that we favor the most: the structural error patterns. While the techniques, other than structural analysis, were not strictly necessary to analyze this data set, they would be required in business process model containing IOR-join.

7.3.3. Additional remarks

In this section, we share our observations on aspects of the results that are not directly linked to efficiency, consumability, or coverage.

7.3.3.1. The type of errors found

The majority of the control-flow errors are either a siphon with a P/T handle or a path to the final place with a T/P handle. While the two types of error occur in a similar order of magnitude, we do detect more paths to the final place with a T/P handle, especially in library B.

One explanation for the large number of paths with a T/P handle is the instantiation semantics of IBM WebSphere Business Modeler which allows the use of multiple start nodes. The default semantics of multiple start nodes is to execute concurrently, i.e., they behave as if an AND-split was generating concurrent tokens on each of them. It is possible to specify a different instantiation semantics but it is rarely done. In our data set, it seems that the modelers often intend to use alternative start nodes, but specify it improperly. XOR-joins are used to merge paths originating from start nodes with concurrent semantics, which results in the simple path to the final place with T/P handle [1]. A manual investigation confirmed that more than half of the paths with T/P handle correspond to a structure originating from concurrent start nodes. The two start nodes in the workflow graph illustrated by Fig. 7.1 are a prime example of start nodes that were modeled with an alternative instantiation semantics in mind. Fig. 7.5 shows a slightly more complex example of improperly specified instantiation semantics resulting in a path to the final place with a T/P handle.

Although initially unmarked siphons appear rather easy to spot, and therefore easy to avoid, we detected a few initially unmarked siphons in the data set. Most of these siphons occur in relatively simple business process models such as the example illustrated in Fig. 7.1. Some initially unmarked siphons, such as the one partially illustrated in Fig. 7.6, are part of large and complex business process models and are significantly more difficult to spot without the help of an analysis tool.

7.3.3.2. The choice and usefulness of the techniques

The results of the first two sets of experiments suggests that the complex analyzer described in Sect. 6.3 is too general, i.e., over-engineered, to analyze the business process models created using the IBM WebSphere Business Modeler. In Sect. 7.3.1, we even observed that the computation of the process structure tree is significantly more time

[1] The beginning of the path and the handle are part of the workflow graph modeling the control-flow implied by the instantiation semantics of the business process model. In such a case, the error pattern cannot be fully displayed on the original business process model. To present this special case to the modeler, one can provide an additional description stating that the two start nodes, which are the first elements of the error pattern that are displayed, have a concurrent instantiation semantics.

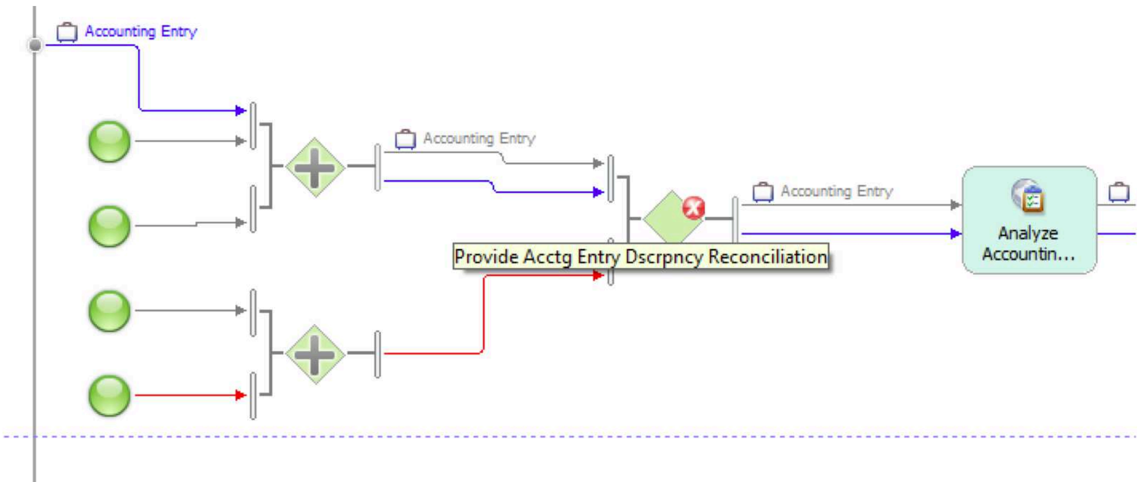


Figure 7.5.: Screenshot of the display of a simple path to the final place with a T/P handle due to a wrongly specified instantiation semantics. The start events in this illustration are instantiated concurrently. Therefore, the blue and the red path are concurrent. However, they join on an XOR-join (empty diamond with error flag) without being properly synchronized.

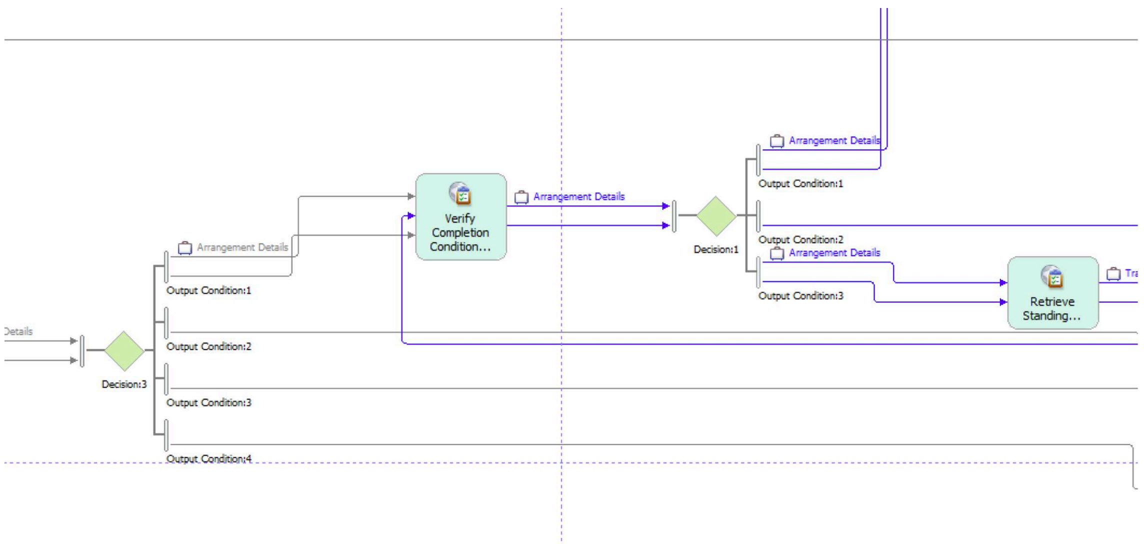


Figure 7.6.: Screenshot of the display of a siphon that is not initially marked displayed in a portion of a large process. The siphon (in blue) is complex and located relatively far from the start events (not in the illustration), which makes it difficult to spot without the help of an analysis tool

consuming than the speed-up that it provides to our structural analysis technique. Building a tailored control-flow analyzer for the IBM WebSphere Business Modeler, we would use a simpler control-flow analyzer relying solely on structural analysis.

Changing slightly the setup, we observed the results change drastically (cf. Table 7.4): the symbolic execution and state space exploration become useful as soon as some IOR-joins are in the process. The process structure tree becomes necessary to be able to analyze fragments in isolation and to profit from the most appropriate technique at hand for each fragment. The use of the process structure also provides further advantages: It allows the modeler to localize and isolate an error within the fragment that contains the error. It also allows the analyzer to detect multiple independents errors per process.

7.3.3.3. The reduction step of structural analysis was not required

During the analysis of more than 50'000 fragments (1350 processes), the structural analysis never had to perform a reduction step, i.e., all the unsound fragments have been identified during the check for non-initially marked siphons or the initial SMD decomposition and, when a fragment was identified as SMD, the rank equation was always satisfied. This confirms our intuition presented at the end of Sect. 3.5.4 that such cases rarely occur in practice.

7.4. Summary

In this chapter, we confirmed that the control-flow analyzer described in Sect. 6.3 satisfies the efficiency, consumability, and coverage requirements formulated in Sect. 1.2.2: On a data set containing more than 1350 industrial business process models, our experiments showed that every process can be analyzed, within a fraction of a second, and, in case of control-flow error, a structural error pattern and a reduced error trace are always delivered as diagnostic information.

As already discussed, the set of techniques chosen to build an analyzer should be tailored to the type of models at hand. We observe that a simplified control-flow analyzer solely based on the structural analysis would be better suited to analyze business precesses modeled using the IBM WebSphere Business Modeler (36). However, symbolic execution could provide complementary diagnostic information for a majority of the error patterns and, in a context where IOR-join can occur, the complete analyzer remains necessary.

Part IV.

Conclusion

8

Conclusion

In this final chapter, we summarize our contributions (Sect. 8.1) and discuss potential directions for future work (Sect. 8.2).

8.1. Summary of the contributions

The problem addressed in this thesis is to detect, locate, understand, and correct control-flow errors in a business process model. We developed two new control-flow analysis techniques and a tool allowing a user without a verification background to perform a control-flow analysis at modeling time. The solution has been designed to be efficient, produce consumable diagnostic information, and cover the constructs of different industrial modeling languages. We also demonstrated a coverage limitation regarding the analysis of business process models containing IOR-joins. We now review these contributions in more details.

8.1.1. Structural analysis and diagnostic information

In Chap. 3, we characterize control-flow errors in workflow nets in terms of structural error patterns: We show that a siphon which does not contain the initial place, a siphon with a P/T handle and a simple path to the final place with a T/P handle are indicative of a control-flow error. We demonstrate the correspondence of the error patterns with control-flow errors by showing that we can derive an error trace of the workflow net from each error pattern. Then, we present a control-flow analysis technique based on this characterization. Upon detecting an error pattern, the analysis technique provides two types of diagnostic information to the modeler:

1. A structural error pattern, which can be shown graphically on the modeling canvas.

2. An error trace, which exemplifies an improper behavior of the process resulting from the error pattern.

Both computations, i.e., the structural analysis and the derivation of the minimal error trace, are in polynomial time.

In summary, in Chap. 3 we present: A structural characterization of soundness and a control-flow analysis technique. This structural control-flow analysis technique, which covers any business process model that can be modeled using a free-choice workflow net, combines efficiency and consumability of the diagnostic information.

8.1.2. Identification of a coverage limitation

In Chap. 4, we prove that business process models containing IOR-logic, and in particular IOR-joins, cannot always be translated into a workflow graph without IOR-join or, equivalently, into free-choice workflow net, i.e., there exist simple processes containing an IOR-join whose synchronization role cannot be implemented using a combination of AND and XOR gateways.

This result shows that any analysis technique, including control-flow analysis technique, based on free-choice Petri nets has coverage issues when it comes to analyzing business process models containing IOR-joins.

While studying the limitations of the translation of a business process model control-flow into a free-choice workflow net, we present two translation techniques allowing us to translate some of the IOR-joins into equivalent free-choice workflow net. We prove application conditions under which these translations provide an equivalent free-choice workflow net.

8.1.3. A symbolic execution for acyclic business process models

In Chap. 5, we present a novel control-flow analysis technique called *symbolic execution*. This symbolic execution allows us to determine under which conditions the elements of an acyclic business process model, which may contain IOR-logic, are executed and to reason about sets of equivalent executions with respect to the elements executed. We use these results to derive the control-flow relationships between the elements of a business process model and, based on these relationships, a control-flow analysis technique. In case of control-flow errors, symbolic execution provides the execution conditions under which the error occurs. These execution conditions allow the modeler to reason about the set of executions in which the error occurs. Based on these execution conditions, we can also derivate, in polynomial time, an error trace exemplifying the control-flow error. Additionally, we show how these conditions can be used by the

user to dismiss spurious errors that arise due to over-approximation of the data-based decisions in the process.

This new technique shows that checking soundness of an acyclic business process containing IOR-logic can be done in quadratic time and space. As a side result, we show that checking another correctness criterion, called week soundness, in acyclic business process models containing IOR-joins is NP-hard.

While the coverage of this symbolic execution is limited to acyclic processes or, as we have discussed in Chap. 6, to acyclic process fragments, this technique allows us to analyze some business process with IOR-logic in quadratic time and to provide a novel type of diagnostic information in terms of execution conditions, which can be combined to an error trace. It also supports the user to identify and dismiss spurious errors.

8.1.4. Building a control-flow analyzer

The structural analysis and symbolic execution techniques presented in this thesis have a polynomial worst case time complexity and provide a valuable diagnostic information however, when used on their own, the techniques cannot analyze all business process models: the structural analysis cannot deal with processes containing IOR-joins and the symbolic execution can only deal with acyclic process models.

In Chap. 6, we describe how to build a hybrid control-flow analyzer capable of analyzing industrial business process models. To satisfy the efficiency, consumability, and coverage requirements discussed in Sect. 1.2.2, this hybrid control-flow analyzer combines the techniques resulting from the previous chapters together with additional state of the art techniques such as, for example, the decomposition of Vanhatalo et al. (56).

We also discuss the user interaction that this hybrid analyzer provides, i.e., the diagnostic information that the techniques deliver, the fix suggestions that can be inferred from the diagnostic information, and the capabilities to dismiss some errors.

Finally, we present the different combination of control-flow analysis techniques that have been implemented during the course of this thesis and transferred to IBM products.

In Chap. 7, we check that the analyzer described in Chap. 6 satisfies our initial requirements of efficiency, consumability, and coverage. We show on a set of 1350 industrial business process models that this analyzer is able to check the soundness of all business process models in a fraction of a second and produce consumable diagnostic information.

8.2. Limitations and directions for future work

In this thesis, we present control-flow analysis techniques allowing a modeler to detect control-flow errors efficiently and providing a useful diagnostic information. However, we did not perform any quantitive user study on the quality of the diagnostic information that the techniques provide. It remains an open question to quantify how useful the provided diagnostic information is.

The modeler also needs to adjust to the structural error patterns provided by the structural analysis and the execution conditions provided by the symbolic execution. The amount time and education required by a modeler to assimilate and fully leverage these new types of diagnostic information is unknown at this point.

We presented potential fix suggestions that can be presented and instantiated by the user in order to correct control-flow errors. A prototype providing tool support for a portion of these fix suggestions together with a concept to rank them has been implemented as part of the master thesis of Deibler (7). However, a complete tool support for error fixing is still lacking. Moreover, while applying one of the suggestions may correct the control-flow error, the pertinence of such a fix is not established: a more thorough analysis would be required to confirm that, when a fix suggestion is applied, the resulting business process model represents the modeler's original intention.

As pointed out during this thesis, some of our results can be useful for data-flow analysis. For example, assuming that we know which data is accessed and written by every activity of a business process model, the control-flow relationships between the activities, which is provided by the symbolic execution, would allows us to detect some data-flow errors. It would be interesting to see how the control-flow analysis techniques presented can be adapted to perform a data-flow analysis and the type of diagnostic information that they could provide.

References

[1] A. Alves et al. Web services business process execution language version 2.0. *OASIS Standard*, 11, 2007. 96, 99

[2] E. Best and P. Thiagarajan. Some classes of live and safe Petri nets. In *Concurrency and nets: Advances in Petri nets*, pages 71–94. Springer-Verlag New York, Inc., 1987. 56

[3] B. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1241, 1976. 6

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. The MIT press, 1990. 127

[5] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri nets for finite transition systems. *IEEE Trans. Computers*, 47(8):859–882, 1998. 99

[6] J. Dehnert and P. Rittgen. Relaxed soundness of business processes. In K. Dittrich, A. Geppert, and M. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 157–170. Springer-Verlag, Berlin, 2001. 29, 126

[7] T. Deibler. Repair-strategien für unsound workflow-graph modelle. Master's thesis, Hochschule in Albstadt-Sigmaringen, 2010. 146, 151, 152, 174

[8] J. Desel. A proof of the rank theorem for extended free choice nets. In K. Jensen, editor, *Application and Theory of Petri Nets 1992*, volume 616 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin Heidelberg, 1992. 8, 54

[9] J. Desel and J. Esparza. *Free choice Petri nets*, volume 40 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, Cambridge, 1995. 8, 17, 20, 36, 41, 54, 103

[10] M. Dumas, W. M. P. van der Aalst, and A. H. ter Hofstede. *Process-aware information systems: bridging people and software through process technology*. John Wiley & Sons, 2005. 4

[11] R. Eshuis and A. Kumar. An integer programming based approach for verification and diagnosis of workflows. *Data Knowl. Eng.*, 69(8):816–835, 2010. 131

[12] J. Esparza. Synthesis rules for Petri nets, and how they lead to new results. *CONCUR'90 Theories of Concurrency: Unification and Extension*, pages 182–198, 1990. 54

[13] J. Esparza. Reduction and synthesis of live and bounded free choice Petri nets. *Information and Computation*, 114(1):50–87, Oct. 1994. 8, 68

[14] J. Esparza and M. Silva. Circuits, handles, bridges and nets. *Advances in Petri nets*, 483:210–242, 1990. 34, 35, 68, 131

[15] J. Esparza and M. Silva. Top-down synthesis of live and bounded free choice nets. In *Advances in Petri Nets 1991*, pages 118–139. Springer, 1991. 54

[16] J. Esparza and M. Silva. A polynomial-time algorithm to decide liveness of bounded free choice nets. *Theoretical Computer Science*, 102(1):185–205, 1992. 50, 55, 57

[17] D. B. et al. OMG Unified Modeling Language (OMG UML), superstructure, v2.1.2. OMG Available Specification OMG Document Number: formal/2007-11-02, OMG Inc., November 2007. 158

[18] D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Instantaneous soundness checking of industrial business process models. In *BPM-09*, volume 5701 of *LNCS*, pages 278–293. Springer, 2009. 9, 136, 143, 154, 158, 159, 161

[19] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data Knowl. Eng.*, 70(5):448–466, 2011. 9, 12, 33, 136, 143, 151, 152, 154, 158, 159, 161

[20] C. Favre. Algorithmic verification of business process models. IBM Research Report RZ 3737, 2009. 29, 136, 143, 154

[21] C. Favre. An efficient approach to detect lack of synchronization in acyclic workflow graphs. In *ZEUS*, volume 563 of *CEUR Workshop Proceedings*, pages 57–64, 2010. 12, 122

[22] C. Favre, D. Fahland, and H. Völzer. The relationship between workflow graphs and free-choice workflow nets. *Information Systems*, volume 47, 2015. 12, 25, 26, 72

[23] C. Favre, Z. Feldman, B. Gfeller, T. Gschwind, J. Koehler, J. M. Küster, O. Maistrenko, A. Marinescu, B. Srivastava, and H. Völzer. A business process services portal (rz3782). Technical report, IBM Research, 2009. 12

[24] C. Favre, T. Gschwind, J. Koehler, W. Kleinöder, A. Maystrenko, K. Muhidini, H. Völzer, and J. Wong. Faster and better business process modeling with the IBM pattern-based process model accelerators. In *BPM (Demos)*, 2009. 12, 153

[25] C. Favre and H. Völzer. Control-flow translation from BPMN 2.0 to workflow graphs. Technical report, 2010. 136

[26] C. Favre and H. Völzer. Symbolic execution of acyclic workflow graphs. In *BPM*, volume 6336 of *LNCS*, pages 260–275. Springer, 2010. 12, 77, 78, 99, 102

[27] C. Favre and H. Völzer. The difficulty of replacing an inclusive OR-join. In *BPM*, LNCS. Springer, 2012. 12, 72

[28] C. Favre and H. Völzer. Computer-implemented method, computer program product and system for analyzing a control-flow in a business process model. patent, Mar. 21 2013. Publication number: US20130073334 A1, Also published as US20130144680 and WO2011148319A1. 12, 102

[29] Gartner. Hype cycle for business process management, 2013. G00246931. 3

[30] C. Gierds and J. Sürmeli, editors. *Proceedings of the 2nd Central-European Workshop on Services and their Composition, ZEUS 2010, Berlin, Germany, February 25–26, 2010*, volume 563 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010. 102

[31] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988. 68, 132

[32] B. Grahlmann. The PEP tool. In *CAV '97*, LNCS 1254, pages 440–443. Springer, 1997. 154

[33] J. Gross and J. Yellen. *Graph Theory and its Applications*. CRC press, 2006. 121

[34] T. Gschwind, J. Koehler, J. Wong, C. Favre, W. Kleinoeder, A. Maystrenko, and K. Muhidini. IBM pattern-based process model accelerators for WebSphere Business Modeler. *IBM developerWorks*, 2009. 12, 153

[35] IBM Software. IBM Business Process Manager, `http://www-03.ibm.com/software/products/en/business-process-manager-advanced`. 12, 146

[36] IBM Software. IBM WebSphere Business Modeler, `http://www-03.ibm.com/software/products/en/modeler-advanced`. 12, 152, 153, 158, 168

[37] IBM Software. IBM WebSphere Integration Developer, `http://www-01.ibm.com/software/integration/wid/`. 12, 153

[38] IBM Software. IBM WebSphere Process Server, `http://www-01.ibm.com/software/integration/wps/`. 12, 153

[39] D. Jordan and J. Evdemon. Web services business process execution language version 2.0. Oasis standard, OASIS, April 2007. 7, 23, 72, 131, 136, 143, 145, 153

[40] P. Kemper. Linear time algorithm to find a minimal deadlock in a strongly connected free-choice net. In M. A. Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 319–338. Springer-Verlag, Berlin, 1993. 57, 68

[41] P. Kemper and F. Bause. An efficient polynomial-time algorithm to decide liveness and boundedness of free-choice nets. *Application and Theory of Petri Nets 1992*, pages 263–278, 1992. 37, 50, 54, 56, 57, 68

[42] B. Kiepuszewski, A. ter Hofstede, and W. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003. 85, 141

[43] A. Kovalyov and J. Esparza. A polynomial algorithm to compute the concurrency relation of free-choice signal transition graphs. In *WODES*, pages 1–6. IEE, 1996. 132

[44] L. M. Kristensen, K. Schmidt, and A. Valmari. Question-guided stubborn set methods for state properties. *Formal Methods in System Design*, 29(3):215–251, Nov. 2006. 9, 154

[45] D. C. Lay. *Linear Algebra and its Applications. 2000.* Addison-Wesley/Longman, New York/London. 59

[46] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979. 94

[47] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *WS-FM*, volume 4937 of *LNCS*, pages 77–91. Springer-Verlag, 2008. 99, 136

[48] N. Lohmann, O. Kopp, F. Leymann, and W. Reisig. Analyzing BPEL4Chor: Verification and participant synthesis. In *WS-FM 2007*, LNCS 4937, pages 46–60. Springer, 2008. 154

[49] A. Martens. On compatibility of web services. *Petri Net Newsletter*, 65:12–20, 2003. 29

[50] J. Mendling. Empirical studies in process model verification. *T. Petri Nets and Other Models of Concurrency (ToPNoC)*, 2:208–224, 2009. 6, 9, 33

[51] J. Mendling, B. van Dongen, and W. van der Aalst. Getting rid of or-joins and multiple start events in business process models. *Enterprise Information Systems*, 2(4):403–419, 2008. 99

[52] J. Mendling, H. M. W. Verbeek, B. F. van Dongen, W. M. P. van der Aalst, and G. Neumann. Detection and prediction of errors in EPCs of the SAP reference model. *Data Knowl. Eng*, 64(1):312–329, 2008. 8, 155

[53] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. 8, 9, 154, 155

[54] OMG. Business process model and notation (BPMN) FTF beta 2 for version 2.0, OMG document number dtc/10-06-04. Technical report, 2010. 7, 10, 23, 72, 136, 143, 158

[55] C. Ouyang, E. Verbeek, W. Van Der Aalst, S. Breutel, M. Dumas, and A. Ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2):162–198, 2007. 99, 136

[56] A. Polyvyanyy, J. Vanhatalo, and H. Völzer. Simplified computation and generalization of the refined process structure tree. In M. Bravetti and T. Bultan, editors, *WS-FM*, volume 6551 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2010. 9, 72, 173

[57] W. Sadiq and M. E. Orlowska. Analyzing process models using graph reduction techniques. *Inf. Syst*, 25(2):117–134, 2000. 8, 15, 68

[58] C. Schröter, S. Schwoon, and J. Esparza. The Model-Checking Kit. In *ICATPN 2003*, LNCS 2679, pages 463–472. Springer, 2003. 154

[59] Y. Shiloach and Y. Perl. Finding two disjoint paths between two pairs of vertices in a graph. *Journal of the ACM (JACM)*, 25(1):1–9, 1978. 119, 120, 121, 122

[60] B. Silver. *BPMN Method and Style*. Cody-Cassidy Press Aptos, 2009. 7

[61] H. Smith and P. Fingar. Business process management (BPM): The third wave. 2006. 3

[62] R. Soley et al. Model driven architecture. *OMG white paper*, 308:308, 2000. 5

[63] C. Stahl. A Petri net semantics for BPEL. Technical report, 2005. 99, 136

[64] W. M. P. van der Aalst. *A class of Petri nets for modeling and analyzing business processes*. Eindhoven University of Technology, Department of Mathematics and Computing Science, 1995. 8, 21

[65] W. M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66, 1998. 17

[66] W. M. P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. *Lecture Notes in Computer Science*, pages 161–183, 2000. 21, 27, 29, 68, 131, 154

[67] W. M. P. van der Aalst, A. Hirnschall, and H. M. W. E. Verbeek. An alternative way to analyze workflow graphs. In *CAiSE 2002*, volume 2348 of *Lecture Notes in Computer Science*, pages 535–552. Springer, 2002. 8, 154

[68] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003. 72

[69] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, R. S. Mans, A. K. A. de Medeiros, A. Rozinat, V. Rubin, M. Song, H. M. W. E. Verbeek, and A. J. M. M. Weijters. ProM 4.0: Comprehensive support for *real* process analysis. In *ICATPN 2007*, LNCS 4546, pages 484–494. Springer, 2007. 154

[70] W. M. P. van Der Aalst and K. M. van Hee. *Workflow management: models, methods, and systems*. MIT press, 2004. 21

[71] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *CONCUR*, volume 458 of *LNCS*, pages 278–297. Springer, 1990. 73

[72] J. Vanhatalo. *Process Structure Trees: Decomposing a Business Process Model into a Hierarchy of Single-Entry-Single-Exit Fragments*. PhD thesis, Universität Stuttgart, 2009. 138, 139, 141, 148

[73] J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. *Data Knowl. Eng.*, 68(9):793–818, 2009. 9, 137

[74] J. Vanhatalo, H. Völzer, and F. Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. In B. J.

Krämer, K.-J. Lin, and P. Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2007. 8, 9, 72, 139

[75] J. Vanhatalo, H. Völzer, and F. Leymann. Faster and more focused control-flow analysis for business process models through SESE decomposition. In B. J. Krämer, K.-J. Lin, and P. Narasimhan, editors, *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2007. 29

[76] J. Vanhatalo, H. Völzer, F. Leymann, and S. Moser. Automatic Workflow Graph Refactoring and Completion. In *ICSOC*, volume 5364 of *LNCS*, 2008. 77, 99

[77] H. Verbeek, T. Basten, and W. Aalst. Diagnosing workflow processes using woflan. *The Computer Journal*, 44(4):246–279, 2001. 8, 9

[78] H. M. W. Verbeek and W. Aalst. Woflan 2.0: A Petri-net-based workflow diagnosis tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 475–484. Springer-Verlag, Berlin, 2000. 21

[79] H. M. W. E. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing workflow processes using woflan. *Comput. J.*, 44(4):246–279, 2001. 154

[80] H. Völzer. A new semantics for the inclusive converging gateway in safe processes. In *Proceedings of the 8th international conference on Business process management*, BPM'10, pages 294–309, Berlin, Heidelberg, 2010. Springer-Verlag. 23, 72, 96, 98, 99, 139

[81] M. Weidlich, A. Polyvyanyy, J. Mendling, and M. Weske. Efficient computation of causal behavioural profiles using structural decomposition. Technical Report BPT 10, HPI, 2010. 101

[82] K. Wolf. Generating Petri net state spaces. In *PETRI NETS 2007*, volume 4546 of *LNCS*, pages 29–42. Springer, 2007. Invited lecture. 9, 154

[83] M. T. Wynn, H. M. W. Verbeek, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Business process verification–finally a reality! *Business Process Management Journal*, 15(1):74–92, 2009. 8, 77, 98, 155

[84] B. Yang, S. Zheng, and E. Lu. Finding two disjoint paths in a network with normalized α–min-sum objective function. 2005. 121

[85] L. Zerguini. A novel hierarchical method for decomposition and design of workflow models. *J. Integr. Des. Process Sci.*, 8(2):65–74, 2004. 9

[86] M. zur Muehlen. *Workflow-based process controlling: foundation, design, and application of workflow-driven process information systems*, volume 6. Michael zur Muehlen, 2004. 4