Uniform Modularization of Workflow Concerns using UNIFY

Uniform Modularization of Workflow Concerns using UNIFY

Niels Joncheere

A dissertation submitted in fulfillment of the requirements for the award of the degree of Doctor of Science

May 2013

Promotor: Prof. Dr. Viviane Jonckers
Co-promotor: Dr. Ragnhild Van Der Straeten

Vrije Universiteit Brussel
Faculty of Science
Department of Computer Science
Software Languages Lab

Print: Silhouet, Maldegem

© 2013 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@aspeditions.be
www.aspeditions.be

ISBN 978 90 5718 304 1
NUR 989
Legal Deposit D/2013/11.161/069

Abstract

Workflow management systems have become a widely accepted technique for automating processes in many domains. A workflow is created by dividing a process into different activities, and specifying the ordering in which these activities need to be performed. This ordering is called the *control flow perspective*. Current workflow languages allow natively expressing a multitude of *control flow patterns* that are related to the fundamental concepts of sequence, parallelism, choice, or iteration. Like any realistic software application, realistic workflows consist of several *concerns* — parts that are relevant to a particular concept, goal, or purpose — which are combined in order to achieve the desired behavior. The general software engineering notion of *separation of concerns* refers to the ability to identify, encapsulate, and manipulate such concerns in isolation of each other. Separation of concerns is traditionally accomplished by decomposing software into modules, which is associated with benefits regarding development time, product flexibility, and comprehensibility. Current workflow languages lack the means to effectively modularize workflow concerns, especially when these concerns are *crosscutting*. Existing work, which includes *aspect-oriented programming* for workflows, offers only partial solutions to this problem. The goal of this dissertation is to improve separation of concerns in workflows by developing a novel, comprehensive approach for modularization of workflow concerns that fills the gaps left by existing work.

We develop a framework named UNIFY, at the heart of which lies a base language meta-model that allows *uniform* modularization of *all* workflow concerns: all workflow concerns, regardless of whether they are regular concerns or crosscutting concerns, are independently specified using the same language construct. We propose a coherent collection of workflow-specific patterns according to which these independently specified concerns can connect to each other, and allow specifying such connections using a connector language meta-model that complements the base language meta-model. The patterns we identify correspond to more expressive, workflow-specific advice types than those supported by aspect-oriented programming languages. Additionally, UNIFY is applicable to several concrete workflow languages, and does not favor a specific implementation strategy.

Because UNIFY's connector mechanism constitutes a novel workflow modularization mechanism, we ensure that the connector mechanism's semantics is precisely described, and that the way in which this semantics is specified fits into the workflow community's existing formal tradition. In order to formalize the aspect-oriented workflow concepts introduced by UNIFY, we employ two complementary formalisms. First, we

augment the static description of UNIFY's workflows as provided by its base language and connector language meta-models with a static semantics for the weaving of UNIFY connectors using the Graph Transformation formalism. Second, we provide a semantics for the operational properties of workflows by proposing a translation to Petri nets, and subsequently extend this semantics to support the operational effects of connectors.

Although UNIFY promotes separation of concerns in workflow languages by offering an improved modularization mechanism, the abstractions offered by existing modularization approaches for workflows, including UNIFY, typically remain at the same level as the base workflow: concerns are implemented using the constructs of the base workflow language, which may not be ideally suited to expressing the concern in question in an efficient, elegant or natural way. Inspired by the benefits of domain-specific languages in general software engineering, we believe that a means of expressing workflow concerns using abstractions that are closer to the concerns' domains can facilitate expressing workflow concerns, and can improve communication with domain experts. Therefore, we develop a methodology for specifying *concern-specific languages* (CSLs) on top of UNIFY, and provide two examples of such CSLs.

Finally, we implement a proof-of-concept of UNIFY, and perform a qualitative validation of the approach. We instantiate the UNIFY framework towards the BPEL and BPMN workflow languages, and perform an initial quantitative validation of UNIFY's performance and scalability.

Samenvatting

Workflow management systemen zijn een algemeen aanvaarde techniek geworden voor het automatiseren van processen in vele domeinen. Een workflow wordt gemaakt door een proces op te delen in verschillende activiteiten en de ordening te specifiëren volgens dewelke deze activiteiten moeten worden uitgevoerd. Deze ordening wordt het *control flow perspectief* genoemd. Huidige workflow-talen laten toe om op een natuurlijke manier een veelheid aan *control flow patronen* uit te drukken die gerelateerd zijn aan de fundamentele concepten van sequentie, parallellisme, keuze of iteratie. Net als elke realistische software-toepassing, bestaan realistische workflows uit verschillende *belangen* — delen die relevant zijn voor een specifiek concept of doel — welke gecombineerd worden teneinde het gewenste gedrag te bereiken. Het begrip *scheiding van belangen* uit de algemene software-ontwikkeling verwijst naar de mogelijkheid om dergelijke belangen te encapsuleren en los van elkaar te manipuleren. Scheiding van belangen wordt traditioneel bereikt door software onder te verdelen in modules, hetgeen geassocieerd wordt met voordelen inzake ontwikkelingstijd, flexibiliteit van het product, en begrijpbaarheid. Huidige workflow-talen ontberen de middelen om workflow-belangen effectief te modulariseren, vooral wanneer deze belangen *doorsnijdend* zijn. Bestaand werk, waaronder *aspectgericht programmeren* voor workflows, biedt enkel gedeeltelijke oplossingen voor dit probleem. Het doel van deze thesis is om scheiding van belangen in workflows te verbeteren door een nieuwe, veelomvattende aanpak voor modularisatie van workflow-belangen te ontwikkelen die de gaten opvult die door bestaand werk werden achtergelaten.

We ontwikkelen een raamwerk genaamd UNIFY, waarvan de kern gevormd wordt door een meta-model voor een basistaal die *uniforme* modularisatie van *alle* workflow-belangen toelaat: alle workflow-belangen, ongeacht ze gewone belangen of doorsnijdende belangen zijn, worden los van elkaar gespecifieerd door middel van hetzelfde taalelement. We stellen een samenhangende verzameling van workflow-specifieke patronen voor volgens dewelke deze los van elkaar gespecifieerde belangen met elkaar verbonden kunnen worden, en laten toe om zulke verbindingen te specifiëren door middel van een meta-model voor een connectortaal dat complementair is aan het meta-model voor de basistaal. De patronen die we identificeren komen overeen met meer expressieve, workflow-specifieke adviestypes dan diegene die ondersteund worden door aspectgerichte programmeertalen. Bovendien is UNIFY toepasbaar op meerdere concrete workflow-talen, en is het niet gericht op één specifieke implementatie-strategie.

Omdat het connectormechanisme van UNIFY een nieuw modularisatiemechanisme

voor workflows vormt, dienen we te verzekeren dat de semantiek van dit connectormechanisme precies beschreven is, en dat de manier waarop deze semantiek beschreven is past bij de bestaande formele traditie binnen de workflow-gemeenschap. Teneinde de aspectgerichte workflow-concepten geïntroduceerd door UNIFY te formaliseren gebruiken we twee complementaire formalismen. Ten eerste vullen we de statische beschrijving van UNIFY-workflows zoals voorzien door de meta-modellen voor de basis- en connectortaal aan met een statische semantiek voor het weven van UNIFY-connectoren door middel van het Graaf-transformatie formalisme. Ten tweede voorzien we een semantiek voor de operationele eigenschappen van workflows door een vertaling naar Petri-netten voor te stellen, en breiden we deze semantiek uit met ondersteuning voor de operationele effecten van connectoren.

Hoewel UNIFY scheiding van belangen in workflow-talen bevordert door een verbeterd modularisatiemechanisme te bieden, blijven de abstracties van zowel bestaande modularisatie-aanpakken voor workflows als UNIFY vaak op hetzelfde niveau als de basisworkflow: belangen worden geïmplementeerd door middel van de taal-elementen van de basistaal, die mogelijk niet ideaal geschikt is voor het uitdrukken van het belang in kwestie op een efficiënte, elegante of natuurlijke manier. Geïnspireerd door de voordelen van domein-specifieke talen in algemene software-ontwikkeling, geloven we dat een manier om workflow-belangen uit te drukken door middel van abstracties die dichter bij het domein van de belangen staan, het uitdrukken van workflow-belangen kan vergemakkelijken en de communicatie met domein-experten kan verbeteren. Daarom ontwikkelen we een methodologie voor het specifiëren van *belang-specifieke talen* bovenop UNIFY, en voorzien we twee voorbeelden van zulke belang-specifieke talen.

Tenslotte implementeren we een proof-of-concept van UNIFY, en voeren we een kwalitatieve validatie van de aanpak uit. We instantiëren het raamwerk naar de BPEL- en BPMN-workflow-talen toe, en voeren een initiële kwantitatieve validatie van UNIFY's performantie en schaalbaarheid uit.

Acknowledgements

Writing this dissertation has only been possible thanks to the support of many people. First and foremost, I would like to thank my promotor, Viviane Jonckers, for offering me the opportunity to pursue a Ph.D., for guiding me during my research, and for proofreading this dissertation.

I am also greatly indebted to my co-promotor, Ragnhild Van Der Straeten, who advised me on a day-to-day basis and with whom I have spent countless hours of fruitful discussion of my research, and finally of this dissertation. In the many years we have shared a desk, first at the SSEL laboratory and later at SOFT, I have not only grown to respect Ragnhild as an outstanding researcher, but also to appreciate her as a most congenial colleague.

This dissertation has also benefited considerably from the feedback of my Ph.D. committee members. Apart from Viviane and Ragnhild, the committee members are Uwe Aßmann, Rubby Casallas, Gilles Geeraerts, Theo D'Hondt, Sven Casteleyn, and Coen De Roover. Many thanks to each of them!

I would also like to thank the many colleagues and former colleagues with whom I have collaborated in the context of my research, especially Wim Vanderperren, Davy Suvée, Mathieu Braem, Bart Verheecke, Dirk Deridder, Bruno De Fraine, Carlos Noguera, Eline Philips, and Sebastian Günther. I made the first LaTeX template for this dissertation by drawing inspiration from Bruno's dissertation, wrote a first version of my introduction after Dirk instructed his own Ph.D. students to do so, and started writing the actual text of this dissertation after a number of interesting talks with Sebastian.

In addition to all these people, I thank the members of the computer science department with whom I have collaborated in the context of education, most notably Wolfgang De Meuter and Andoni Lombide Carreton. Simon De Schutter deserves special mention for taking over some of my teaching commitments during the past months. I am also grateful to the department's secretaries — Brigitte Beyens, Simonne De Schrijver, Lara Mennes, and Lydie Seghers — for their help in many practical and administrative matters, and to those (former) members of the SSEL and SOFT laboratories that I have not yet mentioned, especially to Andy Kellens, Stefan Marr, and Mattias De Wael for the pleasant chats during breaks at work.

Of course, I was not only supported by my colleagues: during evenings, week-ends and holidays, I was glad to spend time with my friends, especially Steven Billens, Ben Van Biesen, Marten Montaine, and Jasper De Bruyn, often while enjoying a nice drink at Jeugdhuis Qw1i in Lennik, at music festivals near Leuven and Hasselt, in the snow-

covered mountains of Val Thorens, or in the green pastures of Saint-Gatien-des-Bois. I would also like to thank the (other) core members of Jeugdhuis Qw1i with whom I have collaborated over the years, as well as the other members of our gastronomic society "Ambrond". And last but not least: thanks to Wim Leo — my drums and percussion teacher, motorcycle driving instructor, and friend — for the weekly hour of teaching and discussion. I will buy him a steak at De Kuiper in Vilvoorde to celebrate the completion of this dissertation.

My final thanks go out to my family: my father Jos Joncheere, my sister Jolien Joncheere, and my mother Françoise Hemelinckx. Their love and support has been invaluable to finishing this dissertation. And finally, I thank the remaining member of my family — my cat Poepoes — for the many hours she spent loafing around my keyboard while I was writing this dissertation. I'm sure she won't mind me blaming any remaining typos in the text on her!

Niels Joncheere
May 2013

Table of Contents

Abstract **v**

Samenvatting **vii**

Acknowledgements **ix**

Table of Contents **xi**

List of Figures **xv**

List of Tables **xviii**

List of Listings **xix**

List of Abbreviations **xxi**

1 Introduction **1**
1.1 Research Context 1
1.2 Research Objectives 3
1.3 Research Methodology 4
1.4 Contributions 6
1.5 Outline 7

2 Context: Separation of Concerns in Workflows **9**
2.1 Workflows 9
2.1.1 History 9
2.1.2 The Workflow Paradigm 10
2.1.3 Terminology 12
2.1.4 Current Application Domains 15
2.1.5 The Business Process Execution Language 17
2.1.6 The Business Process Model and Notation 20
2.1.7 Yet Another Workflow Language 22
2.2 Separation of Concerns 23
2.2.1 General Principles 23
2.2.2 Aspect-Oriented Programming 24

2.3 Separation of Concerns in Workflows . . . 30
2.3.1 The Sub-Workflow Mechanism . . . 31
2.3.2 Aspect-Oriented Programming for Workflows . . . 31
2.4 Summary . . . 33

3 Modularization of Crosscutting Workflow Concerns using PADUS 35
3.1 Context: The WIT-CASE Project . . . 35
3.2 Motivation and Requirements . . . 36
3.3 Language . . . 38
3.3.1 Joinpoint Model and Pointcut Language . . . 38
3.3.2 Advice Model and Language . . . 41
3.3.3 Aspect Module Model . . . 43
3.3.4 Aspect Instantiation and Composition Models . . . 46
3.4 Case Study: The Billing Concern . . . 47
3.5 Architecture and Implementation . . . 48
3.5.1 Architecture . . . 48
3.5.2 Weaver Implementation . . . 49
3.6 The Service Creation Environment . . . 50
3.6.1 Overview . . . 50
3.6.2 Guiding the Service Composition Process . . . 52
3.7 Summary . . . 55

4 Uniform Modularization of Workflow Concerns using UNIFY 57
4.1 Motivation and Requirements . . . 57
4.2 Approach . . . 61
4.3 Base Language . . . 62
4.3.1 Control Flow Perspective . . . 62
4.3.2 Data Perspective . . . 65
4.4 A Coherent Collection of Workflow-Specific Concern Connection Patterns 67
4.4.1 Existing Workflow Patterns . . . 67
4.4.2 Outline of Our Proposal . . . 68
4.4.3 External Concern Connection Patterns . . . 69
4.4.4 Internal Concern Connection Patterns . . . 75
4.4.5 Realization of Concern Connection Patterns in Existing Approaches 76
4.4.6 Conclusions . . . 78
4.5 Connector Mechanism . . . 79
4.5.1 Joinpoint Model and Pointcut Language . . . 79
4.5.2 Advice Model and Language . . . 82
4.5.3 Aspect Module Model . . . 82
4.5.4 Aspect Composition Model . . . 93
4.6 Discussion . . . 97
4.7 Summary . . . 98

5 A Formal Semantics for Aspect-Oriented Workflow Languages 101
5.1 Motivation and Requirements . . . 101

5.2 Towards a Formalization of Aspect-Oriented Workflow Languages 102
5.3 Graph Transformation Formalization of Connectors 103
5.3.1 The Graph Transformation Formalism 103
5.3.2 Graph Transformation Rules 105
5.3.3 Analysis 112
5.4 Petri Net Formalization of Concerns and Connectors 117
5.4.1 Existing Petri Net Formalizations of Workflows 117
5.4.2 Petri Net Formalization of Concerns 119
5.4.3 Petri Net Formalization of Connectors 126
5.4.4 Analysis 138
5.5 Summary 148

6 Modularizing Workflow Concerns using Concern-Specific Languages 151
6.1 Motivation 151
6.2 From Domain-Specific to Concern-Specific Languages 152
6.3 General Methodology 153
6.4 The Access Control CSL 155
6.4.1 Language 155
6.4.2 Translation to UNIFY 157
6.5 The Parental Control CSL 159
6.5.1 Language 159
6.5.2 Translation to UNIFY 161
6.6 Discussion 164
6.7 Summary 166

7 Implementation and Validation of UNIFY 167
7.1 Implementation 167
7.1.1 JAVA Implementation of the UNIFY Base Language and Connector Mechanism 169
7.1.2 Instantiations of the UNIFY Framework 169
7.1.3 Connectors and Compositions 170
7.1.4 The UNIFY Weaver 170
7.1.5 The UNIFY Petri Net Engine 171
7.1.6 Concern-Specific Languages 171
7.2 Validation 172
7.2.1 Expressiveness of UNIFY: Basic Connectors 172
7.2.2 Expressiveness of UNIFY: Advanced Connectors 181
7.2.3 Performance and Scalability of UNIFY 188
7.2.4 Discussion 191
7.3 Summary 192

8 Conclusions 195
8.1 Summary and Contributions 195
8.2 Discussion and Future Work 199

A UNIFY Connector Syntax **203**

B Soundness Proof for the After Connector **205**

C Access Control and Parental Control CSL Syntax **209**
C.1 Access Control CSL Syntax 209
C.2 Parental Control CSL Syntax 211

D Generated Parental Control Concerns **213**

Bibliography **217**

List of Figures

2.1 Common flowchart symbols 10
2.2 Relationships between basic concepts (Workflow Management Coalition, 1999) 14
2.3 The business process management lifecycle (van der Aalst et al., 2003) 16
2.4 Screenshot of the Eclipse IDE with an example BPEL process in its XML representation (left) and as visualized by the Eclipse BPEL Designer (right) 19
2.5 Example order handling workflow, expressed using BPMN 21
2.6 Main BPMN modeling elements 22
2.7 Screenshot of the YAWL Editor with a YAWL representation of our example order handling workflow 23
2.8 YAWL language constructs (van der Aalst and ter Hofstede, 2005) 24
2.9 Modularization of two concerns in the implementation of the TOMCAT web server. Figures adapted from Hilsdale et al. (2001). 26

3.1 PADUS weaver architecture 50
3.2 Service Creation Environment architecture 51
3.3 Service Creation Environment screenshot 52
3.4 SCE guideline verification report 54

4.1 Example order handling workflow, expressed using BPMN 59
4.2 The UNIFY base language meta-model 64
4.3 Independently specified workflow concerns 66
4.4 The UNIFY data meta-model for the *no data passing* approach 67
4.5 The “before” concern connection pattern 71
4.6 The “after” concern connection pattern 71
4.7 The “replace” concern connection pattern 72
4.8 The “around” concern connection pattern 72
4.9 The “parallel” concern connection pattern 73
4.10 The “alternative” concern connection pattern 74
4.11 The “iterating” concern connection pattern 74
4.12 The “synchronizing parallel branches” concern connection pattern 76
4.13 The “switching alternative branches” concern connection pattern 77
4.14 The UNIFY connector language meta-model 80
4.15 The augmented UNIFY data meta-model for the *no data passing* approach . . 84

4.16 Example before connector . . . 85
4.17 Example conditional before connector . . . 85
4.18 Example after connector . . . 86
4.19 Example conditional after connector . . . 86
4.20 Example replace connector (inversion of control) . . . 87
4.21 Example replace connector (hierarchical decomposition) . . . 88
4.22 Example around connector . . . 88
4.23 Example conditional around connector . . . 89
4.24 Example parallel connector . . . 89
4.25 Example conditional parallel connector . . . 90
4.26 Example alternative connector . . . 90
4.27 Example iterating connector . . . 91
4.28 Example synchronizing connector . . . 92
4.29 Example switching connector . . . 94

5.1 Screenshot of UNIFY's type graph (bottom) and `Before` rule (top) in AGG . . . 104
5.2 The `Before` graph transformation rule . . . 105
5.3 The `After` graph transformation rule . . . 106
5.4 The `Replace` graph transformation rule . . . 107
5.5 The `Around` graph transformation rule . . . 108
5.6 The `Parallel` graph transformation rule . . . 109
5.7 The `Alternative` graph transformation rule . . . 110
5.8 The `Iterating` graph transformation rule . . . 111
5.9 The `Synchronizing` graph transformation rule . . . 112
5.10 The `Switching` graph transformation rule . . . 113
5.11 Numbers of mutual exclusions between graph transformation rules as computed by AGG . . . 114
5.12 Numbers of causal dependencies between graph transformation rules as computed by AGG . . . 116
5.13 Petri net patterns for workflow primitives (van der Aalst, 1998b) . . . 117
5.14 Example workflow (top) and corresponding Petri net (bottom) (van der Aalst, 1998b) . . . 118
5.15 Expected corresponding Petri net for example workflow of Figure 5.14; note the OR-join pattern at the right of the figure . . . 118
5.16 Construction of a Petri net N as a sequence of two Petri nets N_1 and N_2 according to the mapping $M_c : O_1 \mapsto I_2 = \{\langle o_1^1, i_2^1\rangle, \ldots, \langle o_1^k, i_2^k\rangle\}$ . . . 123
5.17 Mapping from UNIFY base language primitives to Petri net elements . . . 125
5.18 Construction of the Petri net that corresponds to the example workflow at the top of Figure 5.14 . . . 127
5.19 Construction of N_{Wc} by applying C_{before} to N_{Wx} . . . 129
5.20 Construction of N_{Wc} by applying C_{after} to N_{Wx} . . . 130
5.21 Construction of N_{Wc} by applying $C_{replace}$ to N_{Wx} . . . 131
5.22 Construction of N_{Wc} by applying C_{around} to N_{Wx} . . . 132
5.23 Construction of N_{Wc} by applying $C_{parallel}$ to N_{Wx} . . . 133
5.24 Construction of N_{Wc} by applying $C_{alternative}$ to N_{Wx} . . . 135

5.25 Construction of N_{Wc} by applying $C_{iterating}$ to N_{Wx} . . . 136
5.26 Construction of N_{Wc} by applying $C_{synchronizing}$ to N_{Wx} . . . 137
5.27 Construction of N_{Wc} by applying $C_{switching}$ to N_{Wx} . . . 139

6.1 Independently specified workflow concerns . . . 152
6.2 Access Control CSL domain concepts and relations . . . 156
6.3 Generated composite activity for the example access control concern . . . 158
6.4 Parental Control CSL domain concepts and relations . . . 160
6.5 Generated composite activity for the example parental control concern's filtering policy . . . 162
6.6 Generated composite activity for the example parental control concern's deny usage policy . . . 163
6.7 Generated composite activity for the example parental control concern's refer usage policy . . . 163
6.8 Generated composite activity for the example parental control concern's monitoring policy . . . 164

7.1 General architecture of the UNIFY implementation . . . 168
7.2 Measurement of runtime and weaving overhead introduced by UNIFY *after* connectors . . . 189
7.3 Measurement of runtime and weaving overhead introduced by UNIFY *parallel* connectors . . . 191

List of Tables

2.1 Original *control flow patterns* (Russell et al., 2006a) . . . 12
2.2 Atomic activities in BPEL . . . 18
2.3 Structured activities in BPEL . . . 19

3.1 State of the art in AOP for BPEL . . . 38
3.2 Behavioral joinpoints in PADUS . . . 39
3.3 Structural joinpoints in PADUS . . . 39
3.4 Main properties of `invoking` joinpoints . . . 40
3.5 Places where an *in* advice can be used . . . 42
3.6 Evaluation of PADUS . . . 56

4.1 Requirements for UNIFY . . . 62
4.2 Mapping from basic workflow patterns (Russell et al., 2006a) to corresponding UNIFY constructs . . . 65
4.3 Concern connection patterns . . . 78
4.4 Pointcut predicates . . . 81
4.5 Comparison of ASPECTJ, AO4BPEL, Courbis and Finkelstein, PADUS, JASCO, and UNIFY modularization approaches . . . 83
4.6 Overview of interactions between external connectors; the numbers refer to the different kinds of interactions listed in Section 4.5.4. Because the complete table is symmetric around its main diagonal, we only show its upper right half. . . . 95

5.1 Overview of the approach . . . 103

6.1 Mapping from CSL artifacts to UNIFY connectors . . . 161

7.1 Comparison of lines of code required to implement the example concerns in WS-BPEL, AO4BPEL, UNIFY, and UNIFY CSLs . . . 177
7.2 Correlation results for our experiments regarding *after* connectors (cf. Figure 7.2(a)) . . . 190
7.3 Correlation results for our experiments regarding *parallel* connectors (cf. Figure 7.3(a)) . . . 191

List of Listings

2.1 Crosscutting *tracing* concern within a JAVA class . . . 25
2.2 ASPECTJ pointcut that selects all method executions within the `MyClass` class . . . 27
2.3 ASPECTJ advice that performs tracing around each joinpoint selected by the `myMethods()` pointcut . . . 28
2.4 ASPECTJ tracing aspect . . . 29
2.5 An AO4BPEL aspect that logs all invocations of the `SmsService` web service 33
2.6 A *process aspect* in Courbis and Finkelstein's approach that logs all invocations of the `SmsService` web service . . . 34

3.1 Bindings for the `invoking` pointcut predicate . . . 40
3.2 Example pointcut in PADUS . . . 40
3.3 Example pointcut in PROLOG . . . 41
3.4 Pointcut predicates for exposing the context of a joinpoint . . . 41
3.5 An advice that logs the start of all invocations of the `SmsService` web service 43
3.6 An advice that logs the start and end of all invocations of the `SmsService` web service . . . 44
3.7 An advice that adds a fault handler to the `CreateCall` scope . . . 44
3.8 An aspect that logs the start and end of all invocations of the `SmsService` web service . . . 45
3.9 An aspect deployment specification . . . 46
3.10 Aspect defining generic billing concepts . . . 47
3.11 Aspect implementing a fixed fee billing scheme . . . 48
3.12 Aspect implementing a duration billing scheme . . . 49

4.1 An example composition consisting of a base concern and four other concerns that are applied to the base concern using four connectors . . . 97

6.1 Example access control concern . . . 157
6.2 Generated around connector for the example access control concern . . . 158
6.3 Example parental control concern . . . 161
6.4 Generated after, alternative, and parallel connectors for the example parental control concern . . . 162

7.1 `Report` activity for order confirmation as implemented in WS-BPEL 173
7.2 Reporting aspect as implemented in AO4BPEL 174
7.3 Reporting concern as implemented in WS-BPEL for use by UNIFY 176
7.4 UNIFY connector for the reporting concern 177

List of Abbreviations

AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
BNF	Backus–Naur Form
BPEL	Business Process Execution Language
BPEL4WS	Business Process Execution Language for Web Services
BPM	Business Process Management
BPMN	Business Process Modeling Notation (later renamed to Business Process Model and Notation)
CBSD	Component-Based Software Development
CSL	Concern-Specific Language
DSL	Domain-Specific Language
EBNF	Extended Backus–Naur Form
HTTP	Hypertext Transfer Protocol
OCL	Object Constraint Language
OMG	Object Management Group
SCE	Service Creation Environment
SDP	Service Delivery Platform
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
W3C	World Wide Web Consortium
WfMC	Workflow Management Coalition
WfMS	Workflow Management System
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Service Description Language
XML	Extensible Markup Language
XPATH	XML Path Language
XSD	XML SCHEMA Definition
YAWL	Yet Another Workflow Language

Chapter 1

Introduction

1.1 Research Context

Workflow management systems (Workflow Management Coalition, 1999; van der Aalst and van Hee, 2002) have become a widely accepted technique for automating processes in many domains, ranging from high-level business process management (van der Aalst et al., 2003) to low-level web service orchestration (Andrews et al., 2003). A workflow is created by dividing a process into different activities, and specifying the ordering in which these activities need to be performed. This ordering is called the *control flow perspective* (van der Aalst et al., 2000), as it describes how control flows between the activities. Typically, control can be split into several branches and joined at a later time. This allows specifying parallelism and choice. Research on many different facets of the workflow paradigm has flourished during the past decade, resulting in, among others, the identification of a multitude of *control flow patterns* (van der Aalst et al., 2000; Russell et al., 2006a) as well as additional perspectives such as *data* (Russell et al., 2004a), *resources* (Russell et al., 2004b), and *exception handling* (Russell et al., 2006b). Popular current workflow languages include BPEL (Andrews et al., 2003; Jordan et al., 2007), BPMN (White et al., 2004), and YAWL (van der Aalst and ter Hofstede, 2005).

Like any realistic software application, realistic workflows consist of several *concerns* — parts that are relevant to a particular concept, goal, or purpose — which are connected in order to achieve the desired behavior. The general software engineering notion of *separation of concerns* (Dijkstra, 1982) refers to the ability to identify, encapsulate, and manipulate such concerns in isolation of each other. Separation of concerns is traditionally accomplished by decomposing software into modules, which is associated with benefits regarding development time, product flexibility, and comprehensibility (Parnas, 1972). However, many current workflow languages do not allow effectively decomposing workflows into different modules: although most workflow languages allow decomposing workflows into sub-workflows,[1] this mechanism is typically aimed at grouping activities instead of facilitating the independent evolution and reuse of concerns. Moreover, a

[1] The most notable exception being BPEL, which does not allow modularizing workflows into separate sub-workflows (Trickovic, 2005).

workflow can only be decomposed according to one dimension with this construct, and concerns that do not align with this decomposition end up scattered across the workflow and tangled with one another.[2] This lack of effective modularization mechanisms makes it hard to add, maintain, remove, or reuse workflow concerns (Arsanjani et al., 2003; Courbis and Finkelstein, 2004; Charfi and Mezini, 2004; Verheecke et al., 2006). In order to improve separation of concerns in workflows, workflow languages should allow concerns to be specified in isolation of each other.

However, allowing concerns to be specified in isolation of each other is not sufficient: in order to obtain the desired workflow behavior, workflow languages should also provide a means of specifying how workflow concerns are connected to each other. In existing workflow languages, the only kind of connection that is generally supported is the classic sub-workflow pattern:[3] a main workflow explicitly specifies that a sub-workflow is to be executed. The choice of which sub-workflow is to be executed is made at design time, and it is hard to make a different choice afterwards. A mechanism that reduces the coupling between main workflow and sub-workflow is therefore desirable.

A second kind of connection between concerns is useful when concerns *crosscut* other concerns in a workflow: some concerns cannot be modularized cleanly using the sub-workflow decomposition mechanism, because their implementation is spread out over multiple locations in the workflow. The sub-workflow construct does not solve this problem, since sub-workflows are called explicitly from within the main workflow. This makes it especially hard to add, maintain, remove or reuse such crosscutting concerns. This problem has been observed in general aspect-oriented research (Kiczales et al., 1997). Aspect-oriented extensions to BPEL, such as AO4BPEL (Charfi and Mezini, 2004) and the approach by Courbis and Finkelstein (2005a), allow specifying crosscutting concerns in separate *aspects*. An aspect allows specifying that a certain workflow fragment, called an *advice*, is to be executed *before*, *after*, or *around* a specified set of activities in the base workflow.[4] However, these aspect-oriented extensions use a new language construct for specifying crosscutting concerns, i.e., aspects. This means that concerns which are specified using the aspect construct can only be reused as an aspect, and not as a sub-workflow. On the other hand, concerns which are specified using the sub-workflow construct can only be reused as a sub-workflow, and not as an aspect. Furthermore, AO4BPEL and the approach by Courbis and Finkelstein combine the specification of a crosscutting concern's behavior and the specification of its connection logic in the same aspect construct, which precludes reuse of the behavior in situations where different connection logic is required. An approach that allows modularizing all concerns' behav-

[2] This problem, which is encountered in general software engineering, has been called the *tyranny of the dominant decomposition* by Tarr et al. (1999).

[3] For example, La Rosa et al. (2011b) report on 11 current workflow languages' support for various kinds of modularization. Most workflow languages only support *duplication* (duplicating workflow elements if they point to the same conceptual definition), *vertical modularization* (i.e., the sub-workflow pattern) and *horizontal modularization* (concurrently executing workflows that invoke each other). If one disregards the specific concern of exception handling, none of the languages natively supports *orthogonal modularization* (independent modeling of crosscutting concerns).

[4] In aspect-oriented terminology, these locations are called *joinpoints*, which are typically selected using declarative *pointcut* expressions.

ior using the same language construct, and specifying connection logic separate from this behavior, would remedy these problems.

In addition to introducing a new, specialized language construct for the modularization of crosscutting concerns, existing aspect-oriented extensions only support the basic concern connection patterns (*before*, *after*, or *around*) that were identified in general aspect-oriented research, and do not sufficiently consider the specific characteristics of the workflow paradigm (Braem et al., 2006c; Joncheere and Van Der Straeten, 2011b). They lack support for other patterns such as parallelism and choice. Furthermore, it is completely impossible to specify more advanced connections between concerns, e.g., specifying that a certain concern is to be executed as a synchronization point between two parallel branches of another concern. Workflow languages would benefit from support for such workflow-specific concern connection patterns.

Finally, the abstractions offered by existing modularization approaches — both in general software engineering and in the domain of workflows — typically remain at the same level as the base language: concerns are implemented using the constructs of the base language, which may not be ideally suited for expressing the concern in question. Although existing aspect-oriented extensions improve separation of concerns, they introduce additional complexity in the implementation of a workflow. This complexity must be bridged in order to communicate the implementation of a workflow to the domain experts who identified the process that is automated by the workflow. Furthermore, implementing new concerns requires detailed knowledge of how the relevant domain concepts are represented in the existing workflow implementation. Inspired by the benefits of domain-specific languages in software engineering (cf. van Deursen et al., 2000, for a general introduction to and an annotated bibliography of domain-specific languages), we believe that a means of expressing workflow concerns using abstractions that are close to the concerns' domains can facilitate expressing workflow concerns, and can improve communication with domain experts.

1.2 Research Objectives

The goal of this dissertation is to improve separation of concerns in workflows by developing a novel, comprehensive approach that fills the gaps left by existing work. The main requirements for this approach are the following:

- The approach must facilitate the design, evolution, and reusability of individual workflow concerns. We aim to accomplish this by allowing all workflow concerns, be they crosscutting or not, to be specified in isolation of each other, with all of these concerns being specified using the same language construct. These ideas are inspired by *symmetric* aspect-oriented programming approaches (Tarr et al., 1999; Suvée et al., 2006), and we call such an approach a *uniform* approach.

- The approach must provide a way of specifying rich connections between workflow concerns. We aim to accomplish this by identifying a coherent set of concern connection patterns that are relevant in the context of the workflow paradigm, and

subsequently allowing workflow concerns to be connected according to these patters in a way that supports the reusability of the connected concerns. Inspired by component-based software development (Shaw and Garlan, 1996) and some general aspect-oriented approaches (Suvée and Vanderperren, 2003), we advocate specifying the connections between our uniform workflow concerns in separate *connectors*.

- The approach must facilitate the specification of concerns by offering support for *concern-specific* abstractions, i.e., abstractions that are close to a given family of concerns' domain. We aim to accomplish this by enabling the definition of *concern-specific languages* on top of our uniform workflow concerns and connectors, and proposing a methodology to define such languages. These ideas are inspired by the benefits of domain-specific languages in software engineering (van Deursen et al., 2000).

In addition to the above requirements that are directly derived from the research context, we also recognize the following requirements:

- Because the correct execution of workflows is of vital importance to an organization, a long tradition of formal verification of workflows exists. For example, the execution semantics of BPEL has been formalized using Petri nets (Lohmann, 2007), and YAWL has been developed by augmenting high-level Petri nets with additional constructs (thus obtaining *extended workflow nets*; cf. van der Aalst and ter Hofstede, 2005). Because new modularization mechanisms have a significant impact on the semantics of their base language, it is important that our new modularization mechanisms fit into this existing formal tradition.

- Existing aspect-oriented extensions to workflow languages are all targeted at BPEL, and cannot be applied easily to other workflow languages. Each of these extensions also favors a specific implementation technique; for example, AO4BPEL and the approach by Courbis and Finkelstein can only be executed using a dedicated BPEL engine. Although this has the advantage that it facilitates adding dynamic features to the modularization mechanism, it precludes compatibility with existing tool chains. We aim for our approach to be applicable to several existing workflow languages, and to be independent of any specific workflow engine.

1.3 Research Methodology

As can be expected in software languages research, we set out to fulfill our research objectives by iteratively developing new software languages and experimenting with them. This dissertation will therefore describe two proposals for workflow languages of increasing scope. The first proposal should thus be seen as a stepping stone towards our second, final proposal which aims to fully address our research objectives, and which will be the main topic of this dissertation.

Our first experiment in facilitating the effective modularization of workflow concerns has taken place in the specific context of the WIT-CASE project, which studied and validated innovative solutions for the creation, deployment and runtime execution of services on top of a novel Service Delivery Platform, which is the service infrastructure operated by a telecom service provider or network operator. Based on the characteristics of the telecom Service Delivery Platform and the goals of the WIT-CASE project, we have compiled a list of requirements for our initial approach, which has taken the form of an aspect-oriented extension to BPEL. The extension, which is called PADUS (Braem et al., 2006c), improves separation of concerns in BPEL workflows by applying aspect-oriented principles to BPEL: crosscutting concerns can be modularized as separate aspects, which can be applied to certain locations in a workflow. PADUS has a rich joinpoint model, which consists of all BPEL activities, and joinpoints can be selected using a high-level, logic pointcut language. Based on the observation that workflow languages require more advanced advice types than the classic *before*, *after*, and *around* advice types, PADUS introduces the *in* advice type. Aspects are instantiated and applied to a workflow using an explicit deployment construct, which allows specifying precedence among aspects. PADUS is implemented as a source code weaver which ensures full compatibility with the existing BPEL tool chain.

As a second experiment, we have developed a framework named UNIFY (Joncheere et al., 2008; Joncheere and Van Der Straeten, 2011b), which goes beyond the scope of PADUS and aims to fulfill all of the objectives enumerated in Section 1.2. At the heart of UNIFY lies a base language meta-model that allows uniform modularization of workflow concerns. Each workflow concern, be it regular or crosscutting, is independently specified, using a single language construct. In this respect, UNIFY is related to symmetric aspect-oriented approaches such as HYPERJ (Tarr et al., 1999) and FUSEJ (Suvée et al., 2006). We propose a coherent collection of workflow-specific patterns according to which these independently specified concerns can connect to each other, and allow specifying such connections using a connector language meta-model that complements the base language meta-model. The patterns we identify correspond to more expressive, workflow-specific advice types than those supported by general aspect-oriented programming languages or by PADUS.

Because UNIFY's connector mechanism constitutes a novel workflow modularization mechanism, we ensure that the connector mechanism's semantics is precisely described, and that this semantics fits into the workflow community's existing formal tradition. Therefore, we provide a formalization of our approach that is compatible with existing research on this topic within the workflow community (van der Aalst, 1997, 1998a, 2000; van der Aalst et al., 2011), but also addresses the specific notion of connection patterns introduced by UNIFY. In order to formalize the aspect-oriented workflow concepts introduced by UNIFY, we employ two complementary formalisms. First, we augment the static description of UNIFY's workflows as provided by its base language and connector language meta-models with a static semantics for the weaving of UNIFY connectors using the Graph Transformation formalism (Rozenberg, 1997; Ehrig et al., 2006). This facilitates static reasoning over the applicability and effects of connectors, and provides a foundation for implementing a static weaver for UNIFY connectors. Second, we provide a semantics for the operational properties of workflows by proposing a translation

to Petri nets (Petri and Reisig, 2008), and subsequently extend this semantics to support the operational effects of connectors. This allows reasoning on the dynamics of UNIFY workflow compositions, and provides a foundation for implementing a dedicated workflow engine for UNIFY.

With UNIFY, we introduce a novel approach that promotes separation of concerns in workflow languages by offering an improved modularization mechanism. However, the abstractions offered by existing modularization approaches for workflows and UNIFY typically remain at the same level as the base workflow: concerns are implemented using the constructs of the base workflow language, which may not be ideally suited to expressing the concern in question in an efficient, elegant or natural way. In order to facilitate expressing workflow concerns and improve communication with domain experts by enabling the definition of workflow concerns using abstractions that are closer to the concerns' domains, we develop a methodology for specifying *concern-specific languages* (CSLs) on top of UNIFY, and provide two examples of such CSLs, i.e., the *Access Control* and *Parental Control* CSLs, respectively.

Finally, we implement a proof-of-concept of UNIFY in JAVA, and instantiate the framework towards the BPEL and BPMN workflow languages. UNIFY workflows can be executed either on our dedicated Petri net based workflow engine, or on any standard BPEL workflow engine using our source code weaver. We perform a qualitative validation of UNIFY's expressiveness, and perform an initial quantitative validation of UNIFY's performance and scalability.

1.4 Contributions

Our contributions can be summarized as follows:

1. We develop a novel, comprehensive approach for modularizing workflow concerns — UNIFY — which allows modularizing *all* workflow concerns using a single language construct, and is thus a *uniform* approach. At the heart of UNIFY lies a base language meta-model that is compatible with a wide range of existing workflow languages.

2. We propose a number of *concern connection patterns* for workflows — patterns according to which workflow concerns can be connected — that go beyond the classic aspect-oriented patterns by taking into account the specific properties of the workflow paradigm. In UNIFY, these patterns take the form of a *connector mechanism* that allows connecting independently specified workflow concerns according to each of the concern connection patterns. This connector mechanism is defined in terms of UNIFY's base language meta-model.

3. We enable the definition of *concern-specific languages* (CSLs) on top of UNIFY, which facilitate the definition of families of concerns using abstractions that are close to the concerns' domains. We propose a general methodology for CSL development and exemplify the methodology using two example CSLs.

4. We propose a precise semantics for UNIFY. We enable the verification of static properties and implementation of static weaving by proposing a semantics for our concern connection patterns based on the Graph Transformation formalism, and enable the verification of operational properties and implementation of dynamic weaving by proposing a semantics based on Petri nets.

5. We provide a proof-of-concept implementation of UNIFY, which has been implemented using JAVA, and which has been extended towards BPEL and BPMN. In order to promote compatibility with existing tool chains, UNIFY does not impose a modified workflow engine. We perform a qualitative validation of the expressiveness of UNIFY, and perform a quantitative validation of the performance and scalability of UNIFY's source code weaver.

These contributions have been partially presented in several research papers. Our initial ideas on applying aspect-oriented principles to BPEL, which have taken the form of the PADUS language and implementation, have been published in (Braem et al., 2006c). In the context of the WIT-CASE project, we have built a Service Creation Environment (SCE) around PADUS, and have reported on this in (Joncheere et al., 2006; Braem et al., 2006b). We have also conducted some initial experiments in introducing concern-specific languages into the SCE, and have reported on this in (Braem et al., 2006a; Joncheere, 2007). Our initial ideas on our final approach for modularizing workflow concerns — UNIFY — have been published in (Joncheere et al., 2008), while a complete description of the approach has been published in (Joncheere and Van Der Straeten, 2011b). As an addition to the latter research paper, we have written a technical report (Joncheere and Van Der Straeten, 2011a) on our Graph Transformation formalization of UNIFY connectors.

1.5 Outline

The outline of this dissertation is as follows:

Chapter 2: Separation of Concerns in Workflows presents the context of this dissertation. First, we introduce the workflow paradigm by describing its history, main concepts, and terminology, and give an overview of the most well-known workflow languages. Next, we introduce the notion of separation of concerns, and how it is achieved in object-oriented applications using aspect-oriented programming. Finally, we discuss how separation of concerns is currently achieved in workflows by reviewing traditional mechanisms for modularization of workflows as well as aspect-oriented approaches for workflows.

Chapter 3: Modularization of Crosscutting Workflow Concerns using PADUS proposes our first solution to the problem of separation of concerns in workflows. It describes PADUS, an approach which is specifically aimed at modularizing crosscutting concerns in the BPEL workflow language using aspect-oriented programming, and which provides significant improvements on the state of the art in this focused scope.

Chapter 4: Uniform Modularization of Workflow Concerns using UNIFY builds upon the lessons learned from Chapter 3 in order to propose a second solution to the problem of separation of concerns in workflows. It describes UNIFY, a framework for uniform modularization of workflow concerns, which improves on PADUS by addressing a wider range of requirements: UNIFY is aimed at modularizing all workflow concerns (i.e., not only crosscutting ones), providing advice types that recognize the specific characteristics of the workflow paradigm, and supporting multiple workflow languages. We introduce the approach itself, describe the meta-models that lie at the heart of the approach, and propose a collection of concern connection patterns before describing the connector mechanism that implements these patterns. Subsequent chapters explore additional topics related to this approach.

Chapter 5: A Formal Semantics for Aspect-Oriented Workflow Languages provides a formal semantics for the aspect-oriented workflow concepts that were introduced in Chapter 4. This allows us to reason both on static properties and operational properties of workflows. Additionally, the formalization presented in this chapter supports the implementation of the UNIFY framework.

Chapter 6: Modularizing Workflow Concerns using Concern-Specific Languages motivates and introduces the notion of concern-specific languages (CSLs). A CSL allows implementing a family of concerns using constructs that closely correspond to the concepts of the concerns' domain. We propose a general methodology for building CSLs on top of the UNIFY framework, and illustrate this methodology by introducing the *Access Control* and *Parental Control* CSLs.

Chapter 7: Implementation and Validation of UNIFY provides an overview of UNIFY's implementation, and subsequently presents an initial validation of the approach with respect to expressiveness, performance, and scalability.

Chapter 8: Conclusions concludes this dissertation by summarizing our problem statement and contributions, presenting and discussing our conclusions, and identifying future work.

Chapter 2

Context: Separation of Concerns in Workflows

This chapter presents the context of this dissertation. First, we introduce the workflow paradigm by describing its history, main concepts, and terminology. We discuss the two main application domains of workflows, and give an overview of three important workflow languages: BPEL, BPMN, *and* YAWL. *Second, we introduce the notion of separation of concerns, and how it is achieved in object-oriented applications using aspect-oriented programming. Finally, we discuss how separation of concerns is currently achieved in workflows by reviewing traditional mechanisms for modularization of workflows as well as aspect-oriented approaches for workflows.*

2.1 Workflows

2.1.1 History

The notion of business processes has a very long history: Adam Smith already described a business process for manufacturing metal pins in his fundamental book on classical economics (Smith, 1776, Paragraph I.1.3):

> "The important business of making a pin is, in this manner, divided into about eighteen distinct operations, which, in some manufactories, are all performed by distinct hands, though in others the same man will sometimes perform two or three of them."

Indeed, the basic idea of a business process is already evident in this example: identifying the different activities of which a process consists — as well as the ordering of these activities — and assigning each of these activities to some resource. Nevertheless, documenting business processes in a structured way did not become common before the beginning of the 20th century. The first structured notation for documenting processes was introduced in the 1920s by Gilbreth (1922), and was aimed at specifying processes

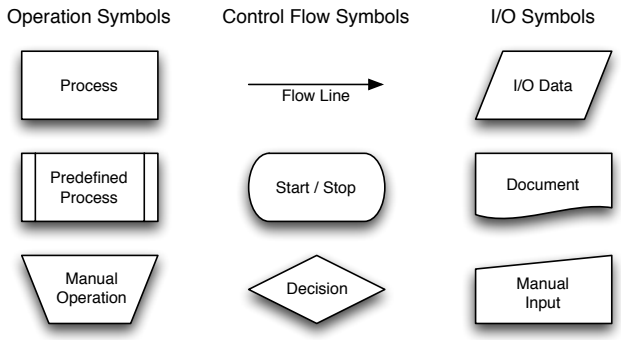


Figure 2.1: Common flowchart symbols

in the manufacturing industry. By the end of the 1950s, several other business process notations became prevalent, such as *Control Flow Diagrams*, *Flowcharts*, and *Functional Flow Block Diagrams*.

For example, Figure 2.1 lists common flowchart symbols. A flowchart has a start symbol and a stop symbol that denote the start and end of a process, respectively. Flow lines represent a process's flow of control, and ultimately connect the start symbol to the stop symbol. In between, operation and input/output symbols may be specified, or the flow of control may be split using decision symbols.

In the 1990s, the domain of *Business Process Reengineering* (Davenport and Short, 1990) constituted an increasing interest in documenting and analyzing business processes in any organization, i.e., not limited to the manufacturing industry. In this context, domain experts started modeling business processes using existing notations such as flowcharts. However, the modeled business processes were still only used as documentation: there was a mismatch between the way in which an organization's business processes were modeled and the way in which software systems supported the organization, which hampered automation of business processes. By the beginning of the 21st century, however, developments in *workflow management* made it possible to automate business processes using executable *workflows*.

2.1.2 The Workflow Paradigm

In an attempt to facilitate the automation of business processes, workflows were introduced as a means of describing business processes in such a way that they can be enacted by a dedicated software system, i.e., a workflow engine (Workflow Management Coalition, 1999). A *workflow language* allows specifying the different units of work (i.e., *activities*) of which a process consists, as well as the ordering of these activities. This is called the *control flow* perspective (Russell et al., 2006a) of the workflow. Typically, workflow languages provide constructs for splitting a workflow's control flow into several branches, and joining these multiple branches afterwards. This allows specifying

parallelism and choice. This focus on the specification of a business process's control flow is what mainly differentiates workflow languages from general-purpose programming languages.

Defined on top of the control flow perspective, the *data* perspective (Russell et al., 2004a) specifies how data is processed by the workflow. For example, a workflow language may allow defining a hierarchy of scopes in which variables may be defined and manipulated by a workflow's activities. Alternatively, a workflow language may allow defining explicit data channels between activities, which are distinct from the workflow's control flow. In addition to the control flow and data perspectives, the *resource* perspective (Russell et al., 2004b) consists of assigning a workflow's work to specific resources, the *exception handling* perspective (Russell et al., 2006b) specifies what should happen when a workflow encounters an error, and the *presentation* perspective (La Rosa et al., 2011a,b) defines how a workflow should be visually represented.

Workflow research typically focuses on the control flow perspective, as the effective specification of business processes' control flow is the main advantage of using workflows over traditional programming paradigms. The data perspective is usually considered in relation to the control flow perspective, while the other perspectives are ancillary (van der Aalst and ter Hofstede, 2005). Within the workflow community, the *Workflow Patterns* initiative (van der Aalst et al., 2012a) aims to provide a conceptual basis for process technology by thoroughly examining the various perspectives that need to be supported by a workflow system using a patterns-based approach. Thus, the following perspectives are examined, with patterns being identified for each of them:

1. The *control flow* perspective (Russell et al., 2006a) defines how a workflow's control flow can be specified. 43 control flow patterns have been identified, which range from five *basic control flow patterns* such as *sequence*, *parallel split* and *exclusive choice*, to advanced control flow patterns related to *advanced branching and synchronization*, *multiple instances*, etc. The first 20 of these patterns, which are known as the original control flow patterns, are listed in Table 2.1.

2. The *data* perspective (Russell et al., 2004a) defines how a workflow's data can be represented and manipulated. 40 data patterns have been identified, which deal with *data visibility*, *data interaction*, *data transfer*, and *data-based routing*.

3. The *resource* perspective (Russell et al., 2004b) defines how resources, i.e., entities that are capable of doing work, are represented and utilized in a workflow. 43 resource patterns have been identified, which deal with various aspects of creating work items and organizing their execution.

4. The *exception handling* perspective (Russell et al., 2006b) defines how exceptions are handled within a workflow. Five possible exception types have been identified. Each exception needs to be handled at certain levels, and may give rise to a certain recovery action. A number of exception handling patterns have been identified, which cover all possible scenarios for the above.

5. The *presentation* perspective (La Rosa et al., 2011a,b) defines how workflows are visually represented. A number of presentation patterns have been identified, which

Basic control flow patterns
WCP-1. Sequence
WCP-2. Parallel split
WCP-3. Synchronization
WCP-4. Exclusive choice
WCP-5. Simple merge
Advanced branching and synchronization patterns
WCP-6. Multi-choice
WCP-7. Structured synchronizing merge
WCP-8. Multi-merge
WCP-9. Structured discriminator
Structural patterns
WCP-10. Arbitrary cycles
WCP-11. Implicit termination
Multiple instances patterns
WCP-12. Multiple instances without synchronization
WCP-13. Multiple instances with *a priori* design-time knowledge
WCP-14. Multiple instances with *a priori* runtime knowledge
WCP-15. Multiple instances without *a priori* runtime knowledge
State-based patterns
WCP-16. Deferred choice
WCP-17. Interleaved parallel routing
WCP-18. Milestone
Cancellation patterns
WCP-19. Cancel activity
WCP-20. Cancel case

Table 2.1: Original *control flow patterns* (Russell et al., 2006a)

constitute different ways of dealing with complexity in workflows when visually representing them.

The above patterns are often used for systematically evaluating the expressiveness of workflow languages with regard to a certain perspective (Vasko and Dustdar, 2004; Mulyar, 2005; Loridan and Anguela Rosell, 2006; van der Aalst et al., 2012b,c).

2.1.3 Terminology

Over the years, many different terms have been used to refer to the various concepts introduced by the workflow paradigm. In the workflow community, efforts have been made to establish a standardized terminology. In this dissertation, we will use the terminology that was standardized by the Workflow Management Coalition (1999), and which is divided in terminology for basic concepts, process concepts, and wider concepts. In Sections 2.1.3.1–2.1.3.3, we list the terms that are most relevant to this dissertation.

2.1.3.1 Basic Concepts

Workflow *The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.*

Workflow Management System (WfMS) *A system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications.*

Business Process *A set of one or more linked procedures or activities which collectively realize a business objective or policy goal.*

Process Definition *The representation of a business process in a form which supports automated manipulation, such as modeling, or enactment by a workflow management system. The process definition consists of a network of activities and their relationships, criteria to indicate the start and termination of the process, and information about the individual activities, such as participants, associated IT applications and data, etc.*

Activity *A description of a piece of work that forms one logical step within a process. An activity may be a manual activity, which does not support computer automation, or a workflow (automated) activity.* A possible synonym for activity is task.

Instance *The representation of a single enactment of a process, or activity within a process, including its associated data. Each instance represents a separate thread of execution of the process or activity.* A possible synonym for process instance is case.

Workflow Participant *A resource which performs the work represented by a workflow activity instance.*

Figure 2.2 (Workflow Management Coalition, 1999) gives a brief overview of most of these basic concepts, as well as the relationships between them. Ironically, the workflow concept is not explicitly present in this figure. Remember that a workflow is the automation of a business process, which is represented in a WfMS by a process definition.

2.1.3.2 Process Concepts

Process *A formalized view of a business process, represented as a coordinated (sequential and/or parallel) set of process activities that are connected in order to achieve a common goal.*

Sub-Process *A process that is enacted or called from another (initiating) process (or subprocess), and which forms part of the overall (initiating) process. Multiple levels of sub-processes may be supported.*

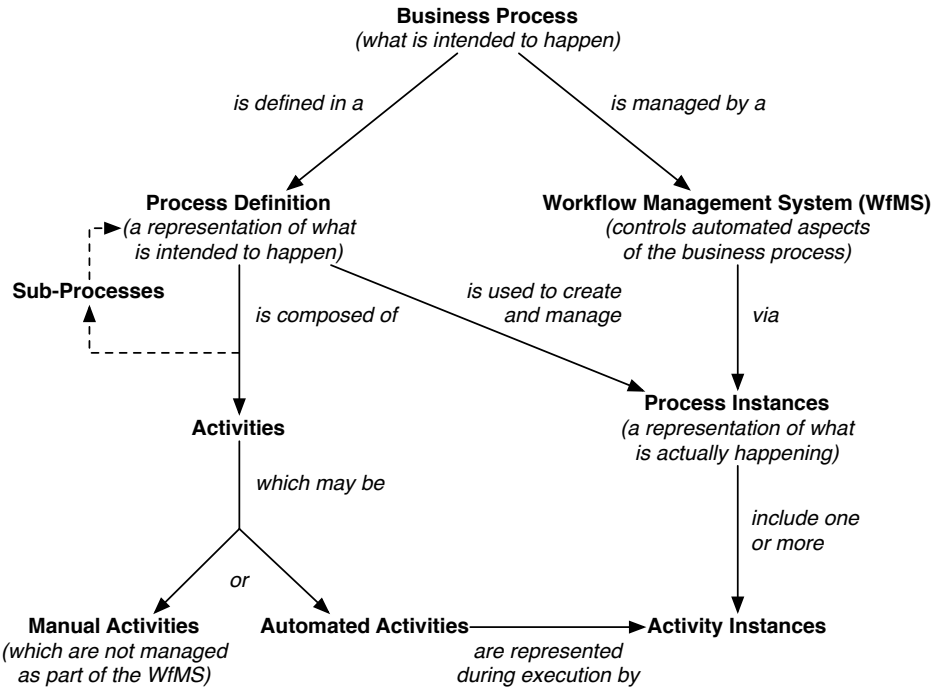


Figure 2.2: Relationships between basic concepts (Workflow Management Coalition, 1999)

AND-Split *A point within the workflow where a single thread of control splits into two or more threads which are executed in parallel within the workflow, allowing multiple activities to be executed simultaneously.*

Note that this definition, as well as the following one, assume that the WfMS is capable of supporting parallelism within a single process instance. In general, AND-splits and AND-joins merely specify concurrency within a process instance.

AND-Join *A point in the workflow where two or more parallel executing activities converge into a single common thread of control.*

OR-Split *A point within the workflow where a single thread of control makes a decision upon which branch to take when encountered with multiple alternative workflow branches.*

OR-Join *A point within the workflow where two or more alternative workflow branches re-converge to a single common activity as the next step within the workflow.*

A workflow in which every split has a corresponding join is called a *structured workflow*; a workflow in which this is not the case is called an *arbitrary workflow*

(Kiepuszewski et al., 2000). Some workflow languages only allow expressing structured workflows.

Iteration *A workflow activity cycle involving the repetitive execution of one or more workflow activities until a condition is met.*

Transition *A point during the execution of a process instance where one activity completes and the thread of control passes to another, which starts.*

2.1.3.3 Wider Concepts

Process Execution *The time period during which the process is operational, with process instances being created and managed.*

Workflow Monitoring *The ability to track and report on workflow events during workflow execution.*

Workflow Engine *A software service or "engine" that provides the runtime execution environment for a process instance.*

2.1.4 Current Application Domains

This section introduces the two main application domains where workflows are currently being used. On the one hand, workflows are used to automate companies' business processes, which are typically coarse-grained, where on the other hand, workflows are used to describe and execute orchestrations of web services, which are typically very fine-grained. These two application domains are discussed in Sections 2.1.4.1 and 2.1.4.2, respectively. In addition to these two application domains, the workflow paradigm is gaining acceptance in some other domains, such as scientific computing (Oinn et al., 2004; Ludäscher et al., 2006) and computer aided engineering (Noesis Solutions, 2006), but these will not be considered further in this dissertation.

2.1.4.1 Business Process Management

Workflows were originally introduced in the context of *business process management* (BPM), which is defined by van der Aalst et al. (2003) as follows:

> "Supporting business processes using methods, techniques, and software to design, enact, control, and analyze operational processes involving humans, organizations, applications, documents and other sources of information."

In this context, domain experts are concerned with modeling the processes used within an enterprise using some modeling notation. This explicit modeling of business processes facilitates analyzing and improving the processes, and verifying whether the processes comply with applicable policies and regulations. Figure 2.3 illustrates the relationship between workflow management and business process management: whereas

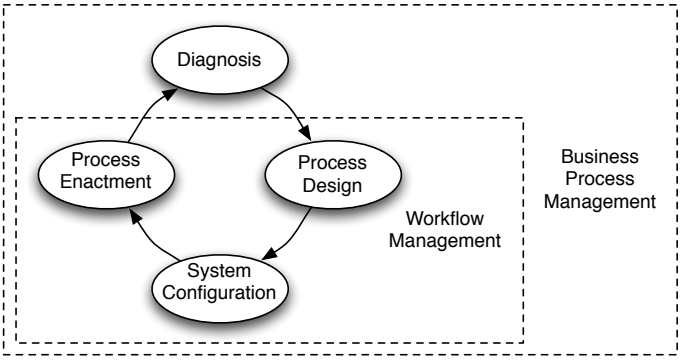


Figure 2.3: The business process management lifecycle (van der Aalst et al., 2003)

BPM focuses on the entire cycle of process design, system configuration, process enactment, and diagnosis (van der Aalst et al., 2003), workflow management only deals with the former three.

Workflow management systems support the automation of business processes by allowing *workflow developers* to translate the (non-executable) business process models defined by domain experts into executable workflows, which are enacted by a workflow engine. The workflow activities in this context often involve humans, e.g., to perform some task or make some decision. Thus, workflows for BPM are typically more high-level than workflows that only deal with software systems, such as workflows for web service orchestration (cf. Section 2.1.4.2).

2.1.4.2 Web Service Orchestration

Services have become a popular means of grouping related software capabilities within an enterprise architecture. *Service-oriented architecture* (SOA) is a paradigm for organizing and utilizing services in a way that promotes the visibility of services, the interaction with services, and the effects of services (MacKenzie et al., 2006). Within a SOA, services are loosely coupled, and the interaction between different services is typically described using explicit *service orchestrations* that map well to the enterprise's business processes. SOAs are commonly implemented using *web services* (Alonso et al., 2004), which are defined by the World Wide Web Consortium (W3C) as follows (Haas and Brown, 2004):

> "A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards."

Web services were developed as a means of facilitating interaction between possibly heterogeneous software systems, by introducing a standard for describing the systems' interfaces (WSDL; cf. Christensen et al., 2001) and a standard for describing the messages exchanged between the systems (SOAP; cf. Box et al., 2000). The actual functionality of a system can be implemented using one of many programming languages with support for web services, and wrappers allow exposing legacy systems as web services.

The advent of web services has greatly facilitated invoking the functionality of external systems over a network. This has given rise to the development of programs that invoke the functionality of several web services in order to offer some new functionality. This is called *orchestration* of web services in common web services terminology. Originally, the orchestration of web services was achieved using general-purpose programming languages, which are not particularly well suited for manipulating the XML data in which web services' messages are expressed: the data manipulation code is interleaved with the core logic of the orchestration, which can make such programs hard to define and understand. This has given rise to dedicated languages for web service orchestration (such as BPEL; cf. Section 2.1.5), which build on earlier work on workflow languages. They allow clearly defining an orchestration's control flow by specifying the ordering in which different activities, such as the invocation of a web service or the manipulation of a variable, must be performed. They are typically XML-based languages, which thus interact well with WSDL, SOAP, and other related standards such as XML SCHEMA (Fallside and Walmsley, 2004). Finally, they facilitate exposing a web service orchestration as a new web service, which can in turn be invoked from other orchestrations.

2.1.5 The Business Process Execution Language

The *Business Process Execution Language* (BPEL) is the de facto standard in workflow languages for web service orchestration. It was originally introduced as the *Business Process Execution Language for Web Services* (BPEL4WS) when IBM and Microsoft decided to merge their existing workflow languages (WSFL and XLANG, respectively) into a new standard, in collaboration with other industrial partners such as BEA Systems, SAP and Siebel Systems (Andrews et al., 2003). The language was renamed to *Web Services Business Process Execution Language* (WS-BPEL) when version 2.0 of its specification was accepted as an OASIS standard (Jordan et al., 2007). Throughout this dissertation, we will simply refer to the language as *Business Process Execution Language* (BPEL) when the differences between the versions are not important.

The goal of BPEL is to provide a standardized way of specifying business processes. A distinction is made between partially specified processes that are not intended to be executed (and which are called *abstract processes* in BPEL terminology) and processes that are fully specified and can thus be executed (and which are called *executable processes*).[1] An abstract process may contain *opaque* tokens (e.g., activities, expressions,

[1] Note that the term *process* in BPEL terminology corresponds to the term *process definition* as defined in the terminology of the Workflow Management Coalition (cf. Section 2.1.3).

Activity	Description
<receive>	Waits for a matching message to arrive
<reply>	Sends a message
<invoke>	Invokes an operation of a certain partner
<assign>	Assigns a certain variable
<throw>	Generates a fault
<exit>	Immediately ends the current process instance
<wait>	Waits for a certain period of time, or until a certain point in time is reached
<empty>	Does nothing
<compensate>	Starts compensation on all inner scopes that have completed successfully
<compensateScope>	Starts compensation on a certain inner scope that has completed successfully
<rethrow>	Rethrows a fault that was caught by the enclosing fault handler
<validate>	Validates the value of a variable against its type

Table 2.2: Atomic activities in BPEL

and attributes) which act as placeholders for the corresponding executable constructs, or they may simply omit parts of the process.

In practice, BPEL is mostly used for specifying executable processes. BPEL is an XML-based language and interacts well with other XML-based web service standards such as WSDL (for specifying web services' interfaces), XML SCHEMA (for specifying XML data), and XPATH (for querying XML data). A typical BPEL process contains the following elements:

- A number of **partner links**, which represent the different partners with which the process will interact, and thus correspond to a number of web services. Each partner link has a type that is defined in the corresponding web service's WSDL interface specification.

- A number of **variables**, which represent the process's global data. Each variable has a type that is defined using XML SCHEMA.

- A **main activity**, which represents the process's workflow. This is typically a structured activity, such as a <sequence>, that composes a number of other activities.

BPEL defines twenty activities, as listed in Tables 2.2 and 2.3. Twelve of these activities have an atomic behavior associated with them, while eight are used to compose a number of activities in a control structure. Typically, a BPEL process's main activity is a <sequence> that starts with a <receive> activity that receives some input and ends with a <reply> activity that returns some output. Between these two activities, other activities are used to compute the output based on the input, while possibly invoking operations of partners.

Figure 2.4 is a screenshot of the Eclipse IDE. The left half of the screen shows an example BPEL process in its XML representation, while the right half of the screen shows the same process as visualized by the Eclipse BPEL Designer (Eclipse Foundation, 2011).

Activity	Description
`<sequence>`	Specifies a sequence of activities
`<if>`	Specifies a conditional structure
`<while>`	Specifies a while-do loop
`<repeatUntil>`	Specifies a repeat-until loop
`<forEach>`	Specifies a foreach loop
`<pick>`	Specifies a choice between several possible activities based on the arrival of a certain message
`<flow>`	Specifies that several activities are to be performed in parallel
`<scope>`	Specifies a nested activity with its own partner links, variables, etc.

Table 2.3: Structured activities in BPEL

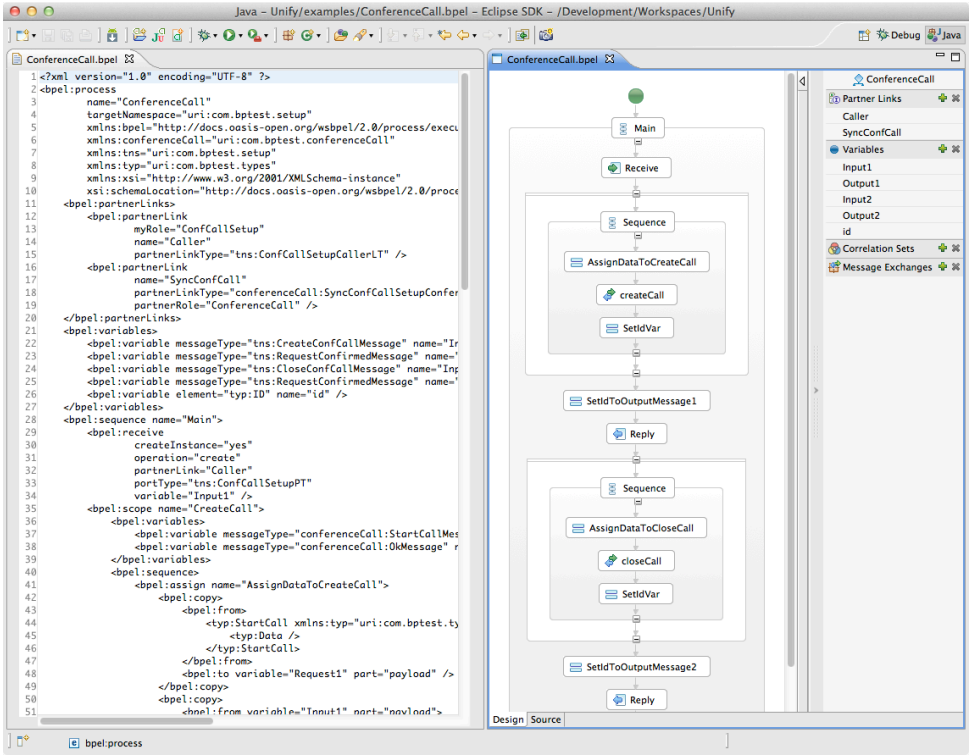


Figure 2.4: Screenshot of the Eclipse IDE with an example BPEL process in its XML representation (left) and as visualized by the Eclipse BPEL Designer (right)

The process was provided by the industrial partner of the WIT-CASE project (cf. Chapter 3), and is used to create and subsequently close a conference call by invoking the operations of an external web service.

BPEL is popular in industrial applications, and is supported by a wide range of workflow engines, such as ACTIVEBPEL (Active Endpoints, 2006) and ODE (Apache Software

Foundation, 2009).

2.1.6 The Business Process Model and Notation

The Business Process Model and Notation (BPMN) was originally introduced as the Business Process Modeling Notation (White et al., 2004), and was thus mainly focused on providing a standardized notation for business processes, though a mapping of BPMN concepts to BPEL4WS was provided as part of the standard. In version 2.0 of its specification (Object Management Group, 2011), the name of the standard was changed to Business Process Model and Notation, in order to reflect its increased focus on providing a complete business process execution model that underlies the existing notation.

BPMN is based on the observation that XML-based workflow languages are optimized for the operation of workflow management systems, and are therefore less suited for direct use by humans, who require a more intuitive means of specifying workflows. In business process management, flowcharts have long been used to facilitate reasoning about business processes by domain experts, but there is a significant technical gap between flowcharts and executable workflow languages that must be bridged in order to visualize workflows using flowcharts, or implement flowcharts using an executable workflow language. BPMN aims to resolve this problem by offering a notation that is well suited for reasoning about business processes by domain experts, but also provides an underlying execution model.

There are four kinds of BPMN models: *private processes*, *public processes*, *choreographies*, and *collaborations*. In the context of this dissertation, only private processes are relevant, as the latter three are not concerned with the complete automation of a single business process, but only with the *interactions* between different business processes. Private processes are either modeled for the purpose of being executed, or modeled for the purpose of documentation. In Figure 2.5, we provide an example order handling workflow that is expressed using BPMN. The main BPMN modeling elements, which are illustrated in Figure 2.6, are the following:

- An **Event** is something that "happens" during the course of a process, and usually has a cause (trigger) or impact (result).
  - A *Start Event* represents the start of a process, which may be triggered, for example, by receiving a message.
  - An *End Event* represents the end of a process, which may result, for example, in sending a message.
  - An *Intermediate Event* represents something that happens while a process is being executed, and allows, for example, waiting for a message to be received.
- An **Activity** is a unit of work to be performed during the course of a process. An activity is either atomic or composite. In BPMN terminology, an atomic activity is a *Task*, whereas common workflow terminology considers *Activity* and *Task* to be synonymous, regardless of whether the unit of work is atomic.

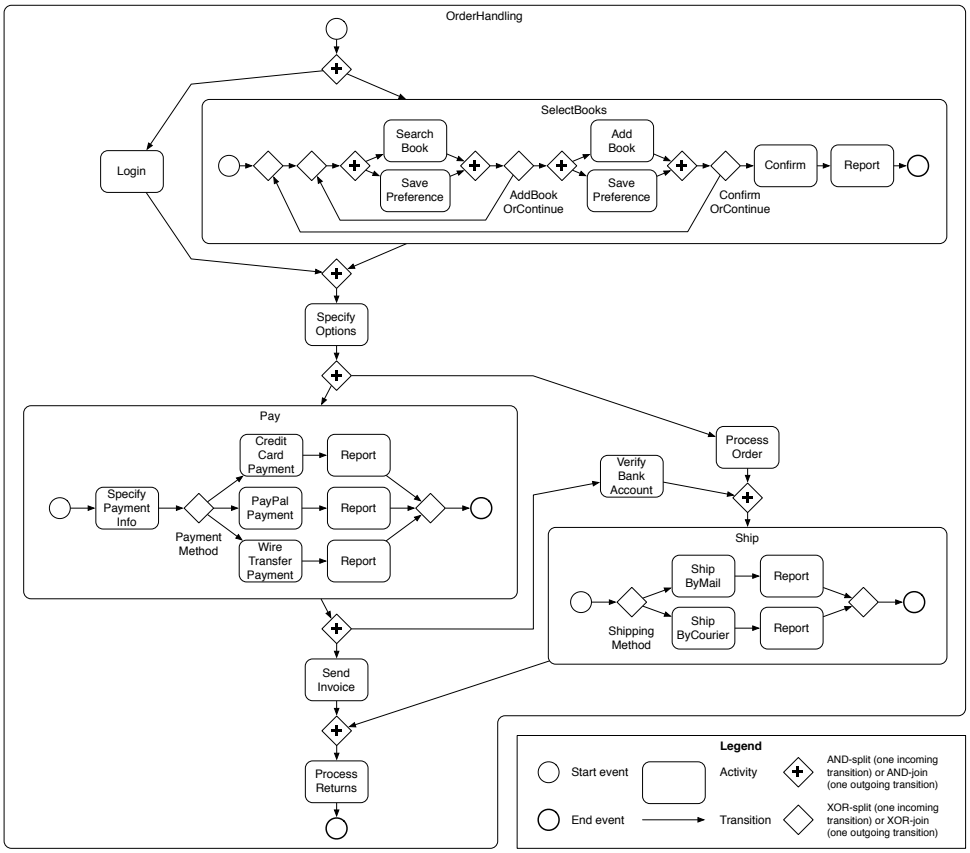


Figure 2.5: Example order handling workflow, expressed using BPMN

- A **Gateway** is used to control the divergence and convergence of a process's control flow, and can thus introduce parallelism or choice. Thus, gateways correspond to various kinds of *Splits* and *Joins* in common workflow terminology.

- A **Sequence Flow** is used to show the order in which a process's activities will be performed. Thus, sequence flows correspond to *Transitions* in common workflow terminology.

Non-executable private processes are specified graphically by combining the above modeling elements in a diagram. For executable private processes, most of the graphical elements must be augmented with textual attributes that specify, for example, which operation is invoked by a service task, or which message triggers a start event.

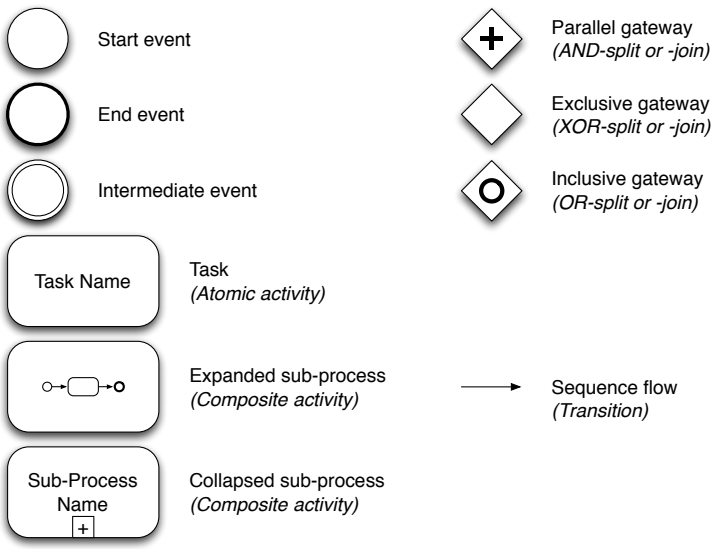


Figure 2.6: Main BPMN modeling elements

2.1.7 Yet Another Workflow Language

Based on the observation that existing workflow languages lacked support for many workflow patterns (cf. Section 2.1.2) and did not offer any formal semantics, van der Aalst and ter Hofstede (2005) introduced *Yet Another Workflow Language* (YAWL) in order to facilitate the use of all workflow patterns in a single workflow language with a solid formal foundation.

Petri nets are well suited as a basis for workflow languages because (1) they have a formal semantics in addition to their graphical nature, (2) they are state-based instead of (just) event-based, and (3) there is an abundance of analysis techniques for Petri nets (van der Aalst, 1998b). *Workflow nets* (WF-nets) (van der Aalst, 1998a) constitute a mapping of workflow management concepts to Petri nets, and thus introduce, among others, notions of process definitions, routing constructs, and activities. Nevertheless, not all workflow patterns can be expressed easily using WF-nets.[2] Therefore, YAWL introduces the notion of *extended workflow nets* (EWF-nets), which extend WF-nets with multiple instances, composite tasks, OR-joins, removal of tokens, and directly connected transitions. A YAWL workflow is an EWF-net that may be hierarchically decomposed into other EWF-nets.

Figure 2.7 is a screenshot of our order handling workflow as defined using the YAWL Editor (YAWL Foundation, 2011). YAWL language constructs, which are illustrated in Figure 2.8, can be divided into two main categories: *Conditions* (which can be inter-

[2] More specifically, *patterns involving multiple instances*, *advanced synchronization patterns*, and *cancellation patterns* (Russell et al., 2006a) require considerable modeling effort.

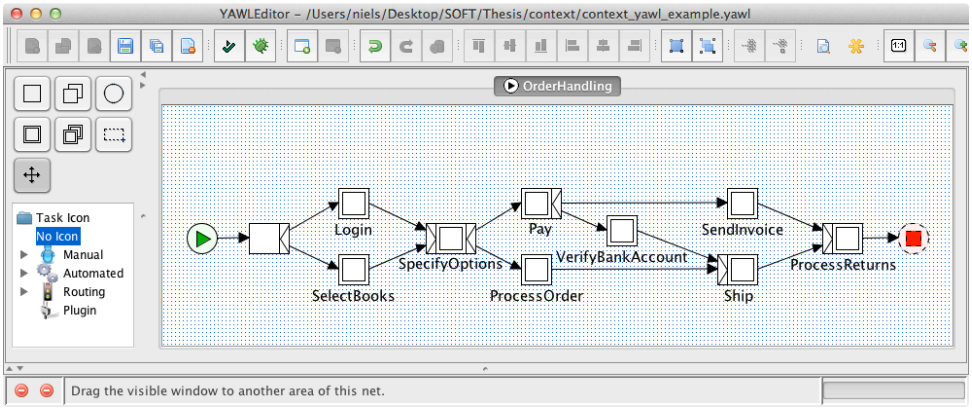


Figure 2.7: Screenshot of the YAWL Editor with a YAWL representation of our example order handling workflow

preted as Petri net *places*) and *Tasks* (which can be interpreted roughly as Petri net *transitions*). Each EWF-net has a single *Input Condition* and a single *Output Condition*, which represent the start and the end of the workflow, respectively. An *Atomic Task* represents an atomic activity, whereas a *Composite Task* represents a composite activity.[3] Both atomic tasks and composite tasks may have multiple instances. Conditions and tasks are connected using directed arcs, which represent the flow of control. This flow of control may be routed using *AND-Split Tasks*, *XOR-Split Tasks*, *OR-Split Tasks*, *AND-Join Tasks*, *XOR-Join Tasks*, and *OR-Join Tasks*. Finally, a task may remove tokens from certain conditions.

2.2 Separation of Concerns

In the previous section, we have introduced the workflow paradigm and have given an overview of three important workflow languages. In this section, we introduce the notion of separation of concerns, and how it is achieved in object-oriented applications using aspect-oriented programming.

2.2.1 General Principles

Most realistic software applications consist of several *concerns* — parts that are relevant to a particular concept, goal, or purpose. *Separation of concerns* (Dijkstra, 1982) is a general software engineering principle that refers to the ability to identify, encapsulate, and manipulate concerns in isolation of each other. Separation of concerns is traditionally accomplished by decomposing software into modules, which is associated with the following benefits (Parnas, 1972):

[3] EWF-net terminology uses the term *Task* as a synonym for *Activity*. This is consistent with common workflow terminology.

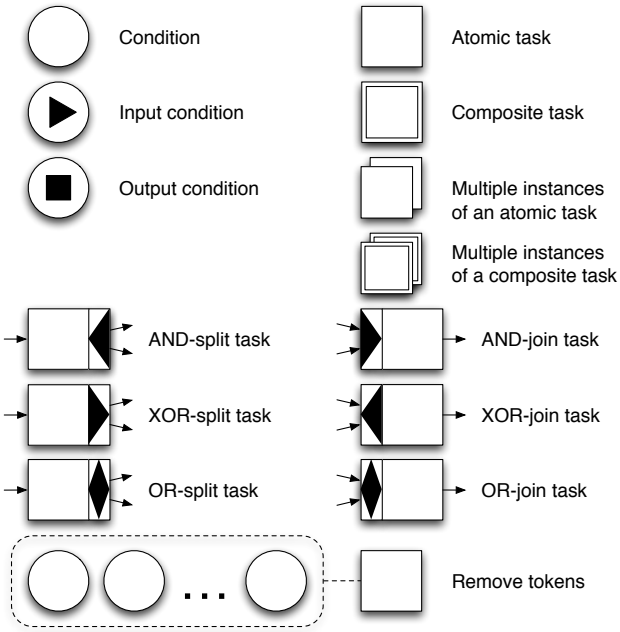


Figure 2.8: YAWL language constructs (van der Aalst and ter Hofstede, 2005)

1. *Managerial — Development time should be shortened because separate groups would work on each module with little need for communication;*

2. *Product flexibility — It should be possible to make drastic changes to one module without a need to change others;*

3. *Comprehensibility — It should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.*

Indeed, these benefits have contributed to the development of many different modularization approaches, which typically allow some sort of hierarchical decomposition of software into modules. However, not all concerns can be modularized by hierarchically decomposing software. More specifically, *crosscutting concerns* pose a problem, as is recognized by the *aspect-oriented programming* community.

2.2.2 Aspect-Oriented Programming

Kiczales et al. (1997) argue that existing modularization approaches cannot cleanly encapsulate concerns that *crosscut* the decomposition hierarchy. According to Tarr et al. (1999), this is due to the "tyranny of the dominant decomposition": using existing modularization approaches, a program can only be modularized according to one dimension

```
1  package com.my_package;
2
3  import org.apache.log4j.Logger;
4  ...
5
6  public class MyClass {
7
8    static Logger logger = Logger.getLogger(MyClass.class);
9
10   public void methodA() {
11
12     logger.debug("Executing methodA()...");
13     ...
14     logger.debug("Executed methodA()");
15   }
16
17   public void methodB() {
18
19     logger.debug("Executing methodB()...");
20     ...
21     logger.debug("Executed methodB()");
22   }
23
24   ...
25 }
```

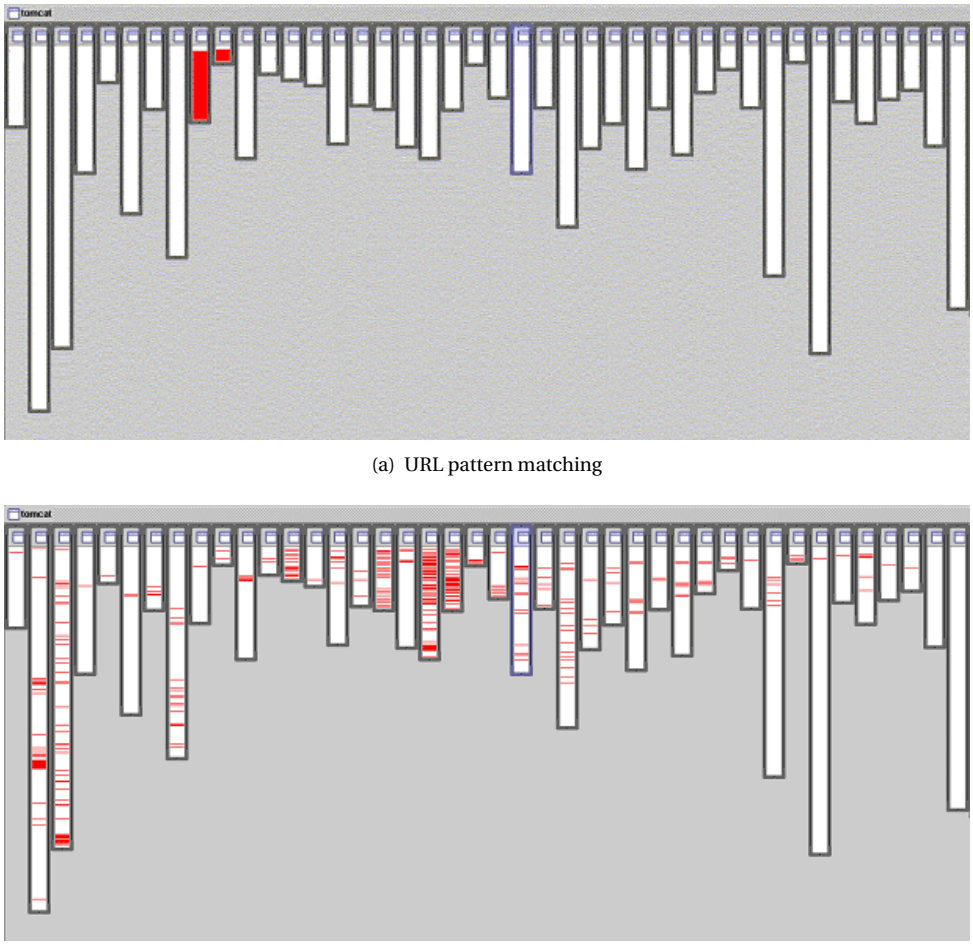
Listing 2.1: Crosscutting *tracing* concern within a JAVA class

at a time, and concerns that do not align with this modularization end up *scattered* across different modules and *tangled* with one another.

For example, most JAVA applications include some kind of tracing scheme that logs information about the applications' execution. This is often accomplished using logging libraries such as LOG4J (Apache Software Foundation, 2002). In all classes whose methods are to be traced, a static variable is defined that initializes a `Logger` (cf. lines 3 and 8 in Listing 2.1). This static variable is then used to log a tracing message at the start (cf. lines 12 and 19) and end (cf. lines 14 and 21) of each method. Clearly, the tracing concern is crosscutting a class's code. Similarly, the concern may crosscut the entire application. This makes it hard to, for example, switch from one logging library to another, as this would require modifying every class and every method where tracing is required. It also means that tracing code must be added to every new class and method as the system evolves. Figure 2.9 illustrates the problem of crosscutting concerns with a less simplified example, by comparing the modularization of the TOMCAT web server's URL pattern matching concern with the (lack of) modularization of its logging concern.

Aspect-oriented programming (AOP) (Kiczales et al., 1997) aims to support programmers in cleanly separating and encapsulating the different concerns of an application. Although several mechanisms can be used to accomplish this goal,[4] the *pointcut/ad-*

[4] Masuhara and Kiczales (2003) have identified four mechanisms: pointcuts and advice, traversal specifications, class composition, and open classes.



(a) URL pattern matching

(b) Logging

Figure 2.9: Modularization of two concerns in the implementation of the TOMCAT web server. Figures adapted from Hilsdale et al. (2001).

```
pointcut myMethods(): within(MyClass) && execution(* *.*(..));
```

Listing 2.2: ASPECTJ pointcut that selects all method executions within the `MyClass` class

vice mechanism is most widely used today. Typically, AOP languages complement a traditional *base language* by allowing crosscutting concerns to be encapsulated in *aspects* that can be applied to a base program. An aspect allows selecting a number of locations (named *joinpoints*) in a base program, and inserting a certain behavior (named *advice*) at these locations. Sets of joinpoints can be selected using a declarative *pointcut language*, and one can specify the position of the inserted behavior relative to the joinpoint (e.g., *before* or *after* the joinpoint). The process of composing the independently modularized base program and aspects in order to support their combined execution is called *weaving*. We will now discuss the different parts of pointcut/advice approaches in more detail. The structure of this discussion is based on the template for describing AOP languages proposed in AOSD-Europe's survey on aspect-oriented programming languages (Brichau and Haupt, 2005).

2.2.2.1 Joinpoint Model and Pointcut Language

The **joinpoint model** of an AOP language defines at which locations an aspect can change a base program. Such joinpoints can be either static (points in the program's structure) or dynamic (runtime events). In ASPECTJ (Kiczales et al., 2001), a popular AOP language that complements the JAVA base language, possible joinpoints are, among others, method call, method execution, constructor call, constructor execution, field get, and field set. For example, ASPECTJ allows us to apply an aspect to the execution of `methodA` or `methodB` of the `MyClass` class defined in Listing 2.1. Because the base program does not need to be aware of the fact that aspect behavior may be inserted at certain locations, the base program is said to be *oblivious* of the aspect behavior. This obliviousness is considered one of the two essential characteristics of AOP mechanisms by Filman and Friedman (2001).

An AOP language allows selecting one or more joinpoints using pointcut expressions. Pointcuts are typically expressed using declarative **pointcut languages**. For example, ASPECTJ pointcuts are logic propositions that are constructed by connecting a number of primitives such as *call* and *execution* for selecting method or constructor calls and executions, respectively, and *get* and *set* for selecting field gets and sets, respectively. Thus, one can select all method executions within the `MyClass` class of Listing 2.1 using the `myMethods()` pointcut that is provided in Listing 2.2. The *execution* primitive's method pattern `* *.*(..)` should be read as: a method with any return type, of any class, with any name, and any number of parameters. This *quantification* over elements of the base program is considered the second essential characteristic of AOP mechanisms by Filman and Friedman (2001).

```
Object around(): myMethods() {

  MethodSignature ms = (MethodSignature)
      thisJoinPointStaticPart.getSignature();
  String mn = ms.getMethod().getName();
  Logger logger = Logger.getLogger(ms.getDeclaringType());
  logger.debug("Executing " + mn + "...");
  Object result = proceed();
  logger.debug("Executed " + mn);
  return result;
}
```

Listing 2.3: ASPECTJ advice that performs tracing around each joinpoint selected by the `myMethods()` pointcut

2.2.2.2 Advice Model and Language

The actual behavior to be inserted at a joinpoint is called the **advice**. The *advice type* defines the position of the inserted behavior relative to the joinpoint. The three classic advice types are *before*, *after*, and *around*, which define that behavior should be inserted immediately before, immediately after, or around a joinpoint, respectively. An advice is associated with a pointcut that selects the joinpoints where the advice is to be inserted. The actual behavior is specified in the advice body. This body is usually specified using the base language. However, in order to allow accessing the (runtime) context in which the advice is executed, or defining the location where the original joinpoint behavior should be executed, the base language may be extended with additional language elements. For example, Listing 2.3 provides an ASPECTJ around advice that retrieves the name of the current joinpoint using the `thisJoinPointStaticPart` context variable, logs an initial tracing message, executes the original joinpoint behavior using the `proceed()` statement, and logs a final tracing message before returning the original joinpoint behavior's result.

2.2.2.3 Aspect Module Model

An **aspect** implements a crosscutting concern by grouping one or more advices, as well as the pointcuts to which they refer. For example, Listing 2.4 shows an ASPECTJ aspect named `TracingAspect` that implements the tracing concern by combining the pointcut of Listing 2.2 (cf. line 8) with the advice of Listing 2.3 (cf. lines 10–19). Thus, the tracing code that was originally crosscutting the `MyClass` class of Listing 2.1 may now be removed from that class, as the `TracingAspect` aspect now encapsulates this code. Similar to JAVA classes, an ASPECTJ aspect may define fields and/or methods, which may be accessed from within the aspect's advice.

In the example of Listing 2.4, the advice that implements the crosscutting behavior is specified in the same file as the pointcut that selects where the crosscutting behavior should be inserted and thus connects the advice to the base program. This ham-

```
1  package com.my_package;
2
3  import org.apache.log4j.Logger;
4  import org.aspectj.lang.reflect.MethodSignature;
5
6  public aspect TracingAspect {
7
8    pointcut myMethods(): within(MyClass) && execution(* *.*(..));
9
10   Object around(): myMethods() {
11
12     MethodSignature ms = (MethodSignature) thisJoinPointStaticPart.getSignature();
13     String mn = ms.getMethod().getName();
14     Logger logger = Logger.getLogger(ms.getDeclaringType());
15     logger.debug("Executing " + mn + "...");
16     Object result = proceed();
17     logger.debug("Executed " + mn);
18     return result;
19   }
20 }
```

Listing 2.4: ASPECTJ tracing aspect

pers reuse of the advice, as it is tightly coupled with the locations to which it is applied. ASPECTJ addresses this issue by allowing the specification of *abstract* aspects that specify abstract pointcuts. Using inheritance, an abstract aspect's behavior can then be reused in different sub-aspects by augmenting the abstract aspect with different concrete pointcuts. A second approach to address this issue is employed by JASCO (Suvée and Vanderperren, 2003), and is inspired by component-based software development (CBSD; cf. Shaw and Garlan, 1996). In this approach, a more conceptual distinction is made between crosscutting behavior and connection logic, by specifying the crosscutting behavior in a separate *aspect bean*, while specifying the connection logic in a separate **connector**. A single aspect bean can then be reused by means of different connectors. Thus, a connector encapsulates the *deployment* of an aspect within an application, which consists of (1) connecting advices to concrete program points, (2) instatiating aspects, (3) configuring aspect instances, and (4) resolving aspect interactions (De Fraine, 2009).

2.2.2.4 Aspect Instantiation Model

When aspects may define a state that is shared between different executions of an advice, the AOP language's aspect instantiation model becomes relevant. The default approach to aspect instantiation is to generate a single aspect instance for any given aspect specification. Such an approach is called implicit instantiation. Optionally, one can specify that multiple aspect instances should be generated. For example, ASPECTJ allows generating one instance per object that contains an execution joinpoint, or one instance per object that contains a call joinpoint. JASCO offers several such options as well, but allows aspect instantiation to be specified in its connectors instead of its aspect beans,

thus allowing aspect beans to be instantiated differently by different connectors.

2.2.2.5 Aspect Composition Model

When multiple aspects are present within the same application, feature interaction problems may arise, most notably when different advices intercept the same joinpoint. For example, one aspect may interfere with the correct execution of another aspect, or the behavior of the application may differ depending on the order in which the aspects are applied. Therefore, AOP languages typically allow specifying *precedence strategies*, which define the order in which aspects are applied when multiple aspects are applicable to the same joinpoint. In ASPECTJ, precedence is specified in the aspects, whereas in JASCO, it is specified in the connectors. In addition to precedence strategies, JASCO allows the definition of *combination strategies*, which allows filtering the list of applicable aspects at a possible joinpoint.

2.2.2.6 Aspect Weaving Model

In order to support the complete execution of an aspect-oriented program, the program's independently modularized aspects should be combined with the base program at a certain point in time. This process is called **weaving**. There are two main approaches to weaving:

- **Static weaving.** In a statically woven approach, the aspect and base code are merged before runtime (e.g., at compile time using a dedicated compiler, or at load time using a dedicated class loader). This merging can take place either at the source code or the byte code level. At runtime, the aspects — like the base code — cannot be redefined or removed, nor can new aspects be added.
- **Dynamic weaving.** A dynamically woven approach uses dedicated techniques (e.g., byte code instrumentation or an aspect-aware VM) to allow weaving at runtime. This allows dynamically adding, removing, and redefining aspects.

The ASPECTJ implementation originally employed static weaving at source code level, but currently weaves byte code at compile or load time. JASCO was designed with dynamic weaving in mind.

Now that we have discussed the origins of separation of concerns and the way in which it is addressed in aspect-oriented programming, we will continue this chapter with discussing separation of concerns in the specific domain of workflows.

2.3 Separation of Concerns in Workflows

Just like any non-trivial software application, a workflow may consist of several concerns, and the ability to identify, encapsulate, and manipulate these in isolation of each other — i.e., separation of concerns — is thus beneficial to the development of the workflow. Most workflow languages allow some separation of concerns by allowing hierarchical decomposition of workflows into sub-workflows. Nevertheless, the problem of crosscutting

concerns arises in this context as well. This has led to the application of aspect-oriented ideas to the workflow paradigm. In Section 2.3.1, we discuss the classic sub-workflow mechanism. In Section 2.3.2, we discuss aspect-oriented programming for workflows.

2.3.1 The Sub-Workflow Mechanism

The notion of sub-processes is an old one: as is illustrated by Figure 2.1, flowcharts already offered a *predefined process* symbol that allows referencing a predefined flowchart from another flowchart. Modern workflow languages offer a sub-workflow mechanism as well:

- BPEL allows hierarchically decomposing processes into *scopes*, which may each define their own partner links, variables, and main activity. However, scopes are defined within the same XML document as the main process, and thus cannot be reused independently. Reuse of BPEL code can be accomplished by exposing a BPEL process as a web service, and then invoking this web service from other BPEL processes, though this introduces significant overhead.

- BPMN allows hierarchically decomposing processes using *sub-processes*, which are essentially composite activities, i.e., activities whose internal details are modeled in the same way as the main process. A sub-process defines a scope that can, for example, be used for exception handling. Sub-processes are defined within the same BPMN diagram as the main processes, but can be reused by other BPMN diagrams using the *call activity* construct.

- YAWL allows hierarchically decomposing workflow specifications using *composite tasks*. Each composite task corresponds to a specific EWF-net, and thus the same EWF-net can be referenced by different workflow specifications.

Various advantages are attributed to the use of this kind of hierarchical decomposition of workflows (Reijers and Mendling, 2008): at design time, sub-workflows facilitate stepwise task refinement, stimulate reuse of workflows, and potentially speed up the (concurrent) development of the overall workflow (Leymann and Roller, 1997; van der Aalst and van Hee, 2002). At runtime, when a workflow is enacted, sub-workflows allow for scaling advantages: each sub-workflow, for example, may be executed by a different workflow engine (Leymann and Roller, 1997). Finally, sub-workflows are believed to facilitate the understanding of complex business processes by various stakeholders (Sharp and McDermott, 2001; Dong and Chen, 2005).

2.3.2 Aspect-Oriented Programming for Workflows

Similar to software expressed using other paradigms, workflows may suffer from poor separation of concerns due to the presence of crosscutting concerns (Arsanjani et al., 2003; Courbis and Finkelstein, 2004; Charfi and Mezini, 2004; Verheecke et al., 2006). For example, the order handling workflow in Figure 2.5 contains reporting activities at

several locations across the workflow. Similarly, such activities may occur across *different* workflows. By applying the ideas of aspect-oriented programming to the workflow paradigm, separation of concerns in workflows can be improved. The two main existing AOP approaches for workflows are introduced below.

2.3.2.1 AO4BPEL

AO4BPEL (Charfi and Mezini, 2004, 2007) is the most well-known aspect-oriented approach for workflows, and adds aspect-oriented capabilities to the BPEL4WS workflow language. AO4BPEL aims to support any BPEL activity as a joinpoint, though the original implementation only supports `<invoke>` and `<reply>` activities. Pointcuts can be specified using XPATH, which is a query language for XML documents, and thus allows selecting activities based on the names and attributes of the corresponding XML elements in the BPEL process definition. Because XPATH is already used within regular BPEL processes to select, for example, parts of structured XML data, it is a natural choice as the pointcut language. On the other hand, this means the pointcut language is strongly tied to the XML representation of the workflow. Advice can be specified using regular BPEL activities, and can be inserted *before*, *after*, or *around* a joinpoint. AO4BPEL performs dynamic weaving using a custom BPEL4WS engine.

Listing 2.5 provides an example AO4BPEL aspect, which logs all invocations of the `SmsService` web service.[5] An aspect is specified using the `<aspect>` element (cf. line 1), which has a `name` attribute for specifying the aspect's name. Optionally, an aspect may introduce a number of partner links (cf. lines 2–4) and variables (cf. lines 5–7) for use by the aspect's advice. In the example, a partner link for a logging service is introduced, as well as a variable that will serve as input for invoking this service. Finally, an aspect has a `<pointcutandadvice>` element (cf. lines 8–23) that groups a `<pointcut>` element and an `<advice>` element. The former (cf. lines 9–11) selects a number of BPEL activities using XPATH, while the latter (cf. lines 12–22) specifies the BPEL code that should be executed at each of the selected activities. In the example, the advice is executed *before* the selected activities, and sends a message that indicates which operation will be invoked to the logging service.

2.3.2.2 Courbis and Finkelstein

Courbis and Finkelstein (2004, 2005a,b) were the first to propose an aspect-oriented approach for workflow languages, but this approach is less well-known than AO4BPEL. Similar to AO4BPEL, the base language is BPEL4WS. Any BPEL activity (i.e., not only `<invoke>` and `<reply>` activities) can be selected as a joinpoint, using XPATH as the pointcut language. Advice can not only be specified using regular BPEL activities (in so-called *process aspects*), but also using JAVA code (in so-called *engine aspects*). In this latter case, the joinpoint activity is exposed to the advice using a JAVA representation of BPEL's abstract syntax. Advice can only be inserted *before* or *after* a joinpoint, i.e., the *around* advice type is not supported. The approach performs dynamic weaving using a custom BPEL4WS engine.

[5] We will use a similar example to illustrate our own approach in Chapter 3 (cf. Listing 3.8).

```
1 <aspect name="logSmsServiceInvocations">
2   <partnerLinks>
3     <partnerLink name="LoggingService" partnerLinkType="LoggingServicePLT"
          myRole="caller" partnerRole="logger" />
4   </partnerLinks>
5   <variables>
6     <variable name="logMessageInput" messageType="ld:logMessageRequest" />
7   </variables>
8   <pointcutandadvice>
9     <pointcut name="smsServiceInvocations">
10      //invoke[@partnerLink="SmsService" and @portType="SmsServicePT"]
11    </pointcut>
12    <advice type="before">
13      <sequence>
14        <assign>
15          <copy>
16            <from expression="concat('Invoking operation ',
                    bpws:getVariableData('ThisJPActivity', 'operation', '/text()'))"
                    />
17            <to variable="logMessageInput" part="payload" />
18          </copy>
19        </assign>
20        <invoke partnerLink="LoggingService" portType="LoggingServicePT"
              operation="logMessage" inputVariable="logMessageInput" />
21      </sequence>
22    </advice>
23  </pointcutandadvice>
24 </aspect>
```

Listing 2.5: An AO4BPEL aspect that logs all invocations of the `SmsService` web service

Listing 2.6 provides an example of a *process aspect* in Courbis and Finkelstein's approach that logs all invocations of the `SmsService` web service. Line 1 indicates the start of the aspect specification, and allows specifying the aspect's name. The `Members` declaration (cf. lines 3–7) allows adding, among others, partner links and variables to the base BPEL process. In the example, a partner link for a logging service is introduced, as well as a variable that will serve as input for invoking this service. The `Pointcuts` declaration (cf. lines 9–12) allows attaching advices to pointcuts according to the *before* or *after* advice types, using XPATH as the pointcut language. Finally, the `Advices` declaration (cf. lines 14–26) specifies the actual BPEL code that should be inserted at the joinpoints. In the example, the BPEL code merely sends the "Invoking operation" message to the logging service, i.e., without the name of the actual operation to be invoked, because it is not possible to access joinpoint information from Courbis and Finkelstein's process aspects (Courbis and Finkelstein, 2005a,b).

2.4 Summary

In this chapter, we present the context of our dissertation. First, we introduce the workflow paradigm by describing its history, main concepts, and terminology. We discuss the

```
1  Workflow aspect logSmsServiceInvocations
2
3  Members {
4
5    add <partnerLink name="LoggingService" partnerLinkType="LoggingServicePLT"
         myRole="caller" partnerRole="logger" />
6    add <variable name="logMessageInput" messageType="ld:logMessageRequest" />
7  }
8
9  Pointcuts {
10
11   before "//invoke[@partnerLink=\"SmsService\" and @portType=\"SmsServicePT\"]"
          insert logSmsServiceInvocations
12 }
13
14 Advices {
15
16   logSmsServiceInvocations
17     <sequence>
18       <assign>
19         <copy>
20           <from expression="'Invoking operation'" />
21           <to variable="logMessageInput" part="payload" />
22         </copy>
23       </assign>
24       <invoke partnerLink="LoggingService" portType="LoggingServicePT"
             operation="logMessage" inputVariable="logMessageInput" />
25     </sequence>
26 }
```

Listing 2.6: A *process aspect* in Courbis and Finkelstein's approach that logs all invocations of the `SmsService` web service

two main application domains of workflows, i.e., business process management and web service orchestration, and give an overview of three important workflow languages, i.e., BPEL, BPMN, and YAWL. Second, we introduce the notion of separation of concerns, and how it is achieved in object-oriented applications using aspect-oriented programming. Finally, we discuss how separation of concerns is currently achieved in workflows by reviewing traditional mechanisms for modularization of workflows as well as aspect-oriented approaches for workflows. Thus, we have set the stage for our own research in the domain of separation of concerns in workflows. In Chapter 3, we develop our own extension to the BPEL workflow language that addresses a number of limitations of the existing aspect-oriented approaches when applying them in the domain of telecom service composition. In Chapter 4, we go beyond the scope of this initial approach by developing a general framework for modularization of workflow concerns, which is the main topic of this dissertation.

Chapter 3

Modularization of Crosscutting Workflow Concerns using PADUS

This chapter presents our first experiments in developing an approach for modularization of concerns in workflows. The approach, which is called PADUS*, is specifically aimed at modularizing crosscutting concerns in the* BPEL *workflow language, and provides significant improvements on the state of the art in this focused scope. Chapter 4 will build on the lessons learned from this chapter in order to develop an approach with a wider scope.*

3.1 Context: The WIT-CASE Project

The research described in this chapter has been performed in the context of the *Workflow Innovations, Technologies and Capabilities for Service Enabling* (WIT-CASE) project, which studied and validated innovative solutions for the creation, deployment and runtime execution of services on top of a novel Service Delivery Platform (SDP), which is the service infrastructure operated by a telecom service provider or network operator. The creation of services within this novel SDP is achieved by means of a Service Creation Environment (SCE), which enables the specification of new services by composing generic service building blocks that are offered by the SDP. This SCE is a graphical environment to be used by domain experts, which translates graphical service compositions into an underlying service composition language. Service compositions expressed using this service composition language can be deployed and executed by a distributed execution environment within the novel SDP. Our research within the WIT-CASE project was mostly aimed at developing the SCE and its underlying service composition language.

It has been established that, similar to traditional programs, service compositions may be burdened by crosscutting concerns that cannot be modularized (Arsanjani et al., 2003; Courbis and Finkelstein, 2004; Charfi and Mezini, 2004; Verheecke et al., 2006). Therefore, the service composition language was conceived to support aspect-oriented mechanisms. Given the increasing popularity of web services technology and dedicated

web service composition languages such as BPEL, both in general industry and in the specific telecom domain, the service composition language was designed as an aspect-oriented extension to BPEL. Thus, we obtain an aspect-oriented workflow language, which is named PADUS, in which crosscutting workflow concerns can be modularized as separate aspects. The outline of this chapter is as follows. Section 3.2 introduces the motivation and requirements for our web service composition language, and compares these requirements to existing approaches. Section 3.3 describes our language along five dimensions: joinpoint model and pointcut language, advice model and language, aspect module model, aspect instantiation model, and aspect composition model. Section 3.4 provides a case study that shows how one can implement the *billing* concern using PADUS. Section 3.5 discusses the architecture and implementation of PADUS, and Section 3.6 describes the SCE that is built on top of PADUS. Section 3.7 concludes this chapter by summarizing our approach.

3.2 Motivation and Requirements

Throughout this chapter, we will illustrate and motivate our approach by providing examples from within the domain of the WIT-CASE project. Typical use cases for an SDP include setting up and executing a multi-party conference call. Such use cases mostly have the same general characteristics. For example, the platform needs to check whether the user is allowed to access the functionality he has requested before providing this functionality (authorization), and the user needs to be billed for his usage according to some billing scheme (billing). The authorization and billing concerns are typically crosscutting. Therefore, an aspect-oriented approach can improve the modularization of web service orchestrations in an SDP. Without support for aspect-orientation, nearly all of the platform's BPEL processes would start with some authorization code before executing their main functionality, and would perform some billing functionality before and/or after certain resources are used. This means that, when some part of the authorization or billing policies is changed, all these processes need to be changed as well. The presence of more than one authorization or billing policy would further complicate this situation. If, on the other hand, the platform would provide support for aspect-orientation, crosscutting concerns such as authorization and billing could be expressed separate from the processes' main functionality in dedicated aspects. If authorization or billing policies would change, this would only require changes to the corresponding aspects, and not to the main processes. If one would like to support more than one authorization or billing policy (e.g., fixed fee billing as well as duration billing), it would suffice to implement an additional aspect. Therefore, we propose an aspect-oriented extension to BPEL, named PADUS, that provides better separation of concerns in BPEL processes by allowing crosscutting concerns to be specified in separate aspects.

Based on the characteristics of the telecom Service Delivery Platform, the goals of the WIT-CASE project, and the state of the art in AOP for BPEL, we have compiled the following list of requirements for our aspect-oriented extension to BPEL:

1. **A rich joinpoint model.** In order to offer the programmer as much freedom as possible when applying aspects to a process, the language should have a joinpoint

model that allows selecting any BPEL activity as a joinpoint.

2. **A high-level pointcut language.** In order to allow the specification of expressive pointcuts that are robust with respect to evolution of the base process, the language should have a pointcut language that abstracts over the XML document tree of BPEL processes and supports reuse of pointcut specifications.

3. **Basic advice types.** In order to be compatible with general aspect-oriented research, the language should support the basic advice types that were introduced there.

4. **Workflow-specific advice types.** The language should support advice types that recognize the workflow paradigm's focus on the control flow perspective, which includes parallelism and choice, in addition to the classic sequential advice types introduced by general aspect-oriented research.

5. **BPEL as the advice language.** In order to promote reuse of process code when adopting the language, it should use BPEL as its advice language. This facilitates the refactoring of process descriptions by moving process code into aspects.

6. **An explicit deployment construct.** In order to prevent undesirable interactions when different aspects apply to the same joinpoint, the language should provide an explicit deployment construct that instantiates and composes the aspects that are applicable to a BPEL process.

7. **An efficient implementation strategy.** In order to prevent undesirable runtime overhead, the language should support an efficient implementation strategy.

8. **Compatibility with the existing tool chain.** In order to facilitate adoption of the language, it should disrupt the existing tool chain as little as possible, and should thus not require a dedicated execution platform.

Table 3.1 gives an overview of the state of the art in AOP for workflows at the time of the WIT-CASE project with respect to these requirements. The most well-known aspect-oriented extensions to BPEL are AO4BPEL (Charfi and Mezini, 2004) and the approach of Courbis and Finkelstein (2005a).

The original implementation of AO4BPEL only allows applying aspects to `<invoke>` and `<reply>` activities, and thus lacks a rich joinpoint model. AO4BPEL's pointcut language is XPATH, which is tightly coupled to the XML document tree and does not support reuse of pointcut specifications. It supports all three basic advice types, but no additional, workflow-specific advice types. It uses BPEL as the advice language, but does not provide an explicit deployment construct. It uses a dedicated BPEL4WS engine as its execution platform, which may not be compatible with existing tool chains.

The approach by Courbis and Finkelstein allows applying aspects to any BPEL activity, and thus provides a rich joinpoint model. Similar to AO4BPEL, XPATH is used as the pointcut language. Only the *before* and *after* advice types are supported: there are no *around* or workflow-specific advice types. Either BPEL or JAVA can be used as the

Requirement	AO4BPEL	Courbis and Finkelstein
A rich joinpoint model	−	+
A high-level pointcut language	−	−
Basic advice types	+	±
Workflow-specific advice types	−	−
BPEL as the advice language	+	+
An explicit deployment construct	−	−
An efficient implementation strategy	±	±
Compatibility with the existing tool chain	−	−

Table 3.1: State of the art in AOP for BPEL

advice language, but the approach does not provide an explicit deployment construct. Like AO4BPEL, the approach uses a dedicated BPEL4WS engine.

3.3 Language

In this section, we present the PADUS language, an aspect-oriented extension to BPEL that aims to overcome BPEL's lack of support for modularization of crosscutting concerns. It allows introducing crosscutting behavior to an existing BPEL process in a modularized way. Developers can augment BPEL processes with additional behavior at specific points during their execution. These points can be selected using a logic pointcut language, and the PADUS weaver can be used to combine the behavior of the base process with the behavior specified in the aspects. Using PADUS, the complexity of the base process can be controlled by specifying crosscutting concerns like authorization and billing in separate aspects.

In order to guide our description of the PADUS language, we follow the same template we used in Chapter 2 to introduce aspect-oriented programming (i.e., the template proposed in AOSD-Europe's survey on aspect-oriented programming languages by Brichau and Haupt, 2005). We describe the language along five dimensions: the joinpoint model and pointcut language (Section 3.3.1), the advice model and language (Section 3.3.2), the aspect module model (Section 3.3.3), the aspect instantiation model (Section 3.3.4), and the aspect composition model (also in Section 3.3.4).

3.3.1 Joinpoint Model and Pointcut Language

Joinpoints are well-defined points during the execution of a program where extra functionality can be inserted. In PADUS, these points correspond to the different activities that are provided by BPEL. There are two kinds of joinpoints: *behavioral joinpoints* correspond to BPEL activities that have an atomic behavior associated with them, as listed in Table 3.2, and *structural joinpoints* correspond to BPEL activities that are used to compose a number of other BPEL activities in a separate structure, as listed in Table 3.3. In essence, PADUS allows selecting any activity in a BPEL process as a joinpoint.

Joinpoint type	Corresponding BPEL activity
receiving	<receive>
replying	<reply>
invoking	<invoke>
assigning	<assign>
throwing	<throw>
exiting	<exit>
waiting	<wait>
doingNothing	<empty>
compensating	<compensate>
compensatingScope	<compensateScope>
rethrowing	<rethrow>
validating	<validating>

Table 3.2: Behavioral joinpoints in PADUS

Joinpoint type	Corresponding BPEL activity
executingSequence	<sequence>
executingIf	<if>
executingWhile	<while>
executingRepeatUntil	<repeatUntil>
executingForEach	<forEach>
executingPick	<pick>
executingFlow	<flow>
executingScope	<scope>

Table 3.3: Structural joinpoints in PADUS

Each joinpoint type is associated with a number of properties relevant to that particular joinpoint type:

- Each joinpoint has a property for each of the attributes of the corresponding BPEL activity. For example, Table 3.4 lists these properties for `invoking` joinpoints.
- Each joinpoint has a *Parent* property that refers to the structured activity or BPEL process in which it is defined.
- Each joinpoint has a *Process* property that refers to the BPEL process in which it is defined.
- Each joinpoint has a (dynamic) *ProcessInstance* property that refers to the current process instance.

A pointcut selects a specific set of joinpoints. Pointcuts are typically used to specify the joinpoints where additional behavior should be inserted. The pointcut language of PADUS is based on logic meta-programming (De Volder, 1998). A pointcut can be seen as a collection of constraints on the types and properties of allowed joinpoints. In addition,

Property name	Type	Description
name	String	The (optional) name of the <invoke> activity
partnerLink	String	The partner link to be used by the <invoke> activity
portType	String	The port type to be used by the <invoke> activity
operation	String	The operation to be invoked
inputVariable	String	The variable containing the request message
outputVariable	String	The variable that should contain the response message

Table 3.4: Main properties of invoking joinpoints

```
% Binds all possible joinpoint properties:
invoking(Joinpoint, Name, PartnerLink, PortType, Operation,
    InputVariable, OutputVariable)

% Does not bind the input and output variable properties:
invoking(Joinpoint, Name, PartnerLink, PortType, Operation)

% Only binds the partner link, port type, and operation properties:
invoking(Joinpoint, PartnerLink, PortType, Operation)
```

Listing 3.1: Bindings for the invoking pointcut predicate

```
<pointcut name="smsServiceSendInvocations(Joinpoint, Operation)"
    pointcut="invoking(Joinpoint, 'SmsService', 'SmsServicePT',
    Operation), startsWith(Operation, 'send')" />
```

Listing 3.2: Example pointcut in PADUS

a pointcut is able to expose certain information (e.g., argument values) so that the advice can exploit this.

The pointcut language defines a predicate for each type of joinpoint. The arguments of the predicate refer to the properties of that specific type of joinpoint. Listing 3.1 lists the exposed bindings of the invoking predicate. Only the first binding — with the most arguments — is really required: the others can easily be written in function of the first one, and are thus mainly offered for convenience.

Pointcuts are logic rules in which PADUS pointcut predicates may be combined with standard PROLOG (Deransart et al., 1996) predicates using conjunctions, disjunctions, and negations, e.g., in order to compare joinpoint properties. By constraining the arguments of a pointcut predicate, a specific set of joinpoints can be selected. Listing 3.2 shows an example pointcut that selects all invoking joinpoints of operations on the SmsServicePT port type of the SmsService partner link, of which the operation starts with "send". The name of a pointcut defines a new pointcut predicate by means of which the pointcut can be used in aspects. Thus, the pointcut of Listing 3.2 is equivalent to the PROLOG rule shown in Listing 3.3.

The pointcut language also offers predicates for constraining or exposing the other

```
smsServiceSendInvocations(Joinpoint, Operation) :- invoking(Joinpoint,
    'SmsService', 'SmsServicePT', Operation), startsWith(Operation,
    'send').
```

Listing 3.3: Example pointcut in PROLOG

```
% Links a joinpoint to the structured activity or process in which it is
    defined:
inStructure(Joinpoint, Structure)

% Links a joinpoint to the process in which it is defined:
inProcess(Joinpoint, Process)

% Links a joinpoint to the process instance in which it occurs:
inProcessInstance(Joinpoint, ProcessInstance)

% Links the name of a variable to its value in a specific process
    instance:
variableValue(ProcessInstance, Name, Value)
```

Listing 3.4: Pointcut predicates for exposing the context of a joinpoint

(possibly dynamic) properties of joinpoints, such as a joinpoint's parent, the process in which a joinpoint is defined, or the process instance in which a joinpoint occurs. Listing 3.4 gives an overview of these predicates.

The use of a logic pointcut language offers significant advantages over more traditional approaches that use XPATH, and which impose a tight coupling between pointcuts and the XML representation of the base BPEL process. Our pointcuts can use the full power of unification of logic variables. Furthermore, since pointcuts are logic rules that cover joinpoints, existing pointcuts can be reused in the definition of other pointcuts. The logic engine supporting our pointcut language also allows writing recursive pointcuts. The predicates offered by the pointcut language have well-chosen names, which clearly express their intension and thus improve readability.

3.3.2 Advice Model and Language

An advice specifies how the behavior at the set of joinpoints defined by a pointcut is to be altered. An advice can either *add* behavior to a joinpoint, or *replace* the original behavior of a joinpoint. PADUS supports four advice types, three of which are the traditional advice types introduced in general aspect-oriented programming by ASPECTJ (Kiczales et al., 2001):

1. A **before** advice adds behavior sequentially before a joinpoint.

2. An **after** advice adds behavior sequentially after a joinpoint.

Joinpoint	Element	Description
All types	source	Specify that the joinpoint is the source of a new link.
	target	Specify that the joinpoint is the target of a new link.
receiving	correlation	Add a new correlation element to the joinpoint.
replying	correlation	Add a new correlation element to the joinpoint.
invoking	correlation	Add a new correlation element to the joinpoint.
	catch	Add a new specific catcher to the joinpoint.
	catchAll	Add a new generic catcher to the joinpoint.
	compensation-Handlers	Add compensation handlers to the joinpoint.
assigning	copy	Add a new copy element to the joinpoint.
executingIf	elseif	Add a new elseif element to the joinpoint.
	else	Add a new else element to the joinpoint.
executingPick	onMessage	Add a new message trigger to the joinpoint.
	onAlarm	Add a new timeout trigger to the joinpoint.
executingFlow	Any activity	Add a new parallel activity to the joinpoint.
	link	Add a new link to the joinpoint.
executingScope	variable	Add a new variable to the joinpoint.
	correlationSet	Add a new correlation set to the joinpoint.
	faultHandlers	Add fault handlers to the joinpoint.
	compensation-Handlers	Add compensation handlers to the joinpoint.
	eventHandlers	Add event handlers to the joinpoint.

Table 3.5: Places where an *in* advice can be used

3. An **around** advice replaces the original behavior of a joinpoint, and (optionally) allows invoking the original behavior using the new <proceed> activity.

These three advice types are well accepted both in general AOP and in AOP for workflows. However, current AOP approaches for workflows (e.g., AO4BPEL) do not offer advice types in addition to these three. Nevertheless, workflow languages' focus on the control flow perspective, with native support for parallelism, may lead to the applicability of more advice types than the three classic types. Therefore, PADUS introduces a fourth advice type:

4. An **in** advice adds behavior inside a joinpoint, and is thus a new advice type introduced by PADUS.

The *in* advice can be used to add new activities to existing structured BPEL activities. For example, an *in* advice can be used to add an extra concurrent activity to an existing <flow> activity. This cannot be achieved using a *before*, *after*, or *around* advice, as these introduce behavior sequentially before and/or after a joinpoint. Additionally, the *in* advice can be used to customize the behavior of certain other BPEL activities, e.g., adding variables to a scope or adding links to an activity. Table 3.5 gives an overview of all the joinpoints where an *in* advice can be used, and explains the advice's use at each of these joinpoints.

Advice code is defined in an XML element that specifies the type of the advice. A logic query that calls a pointcut predicate allows selecting the activities in the BPEL process to which the advice applies. The extra behavior to be inserted is specified using standard

```
1 <before joinpoint="Joinpoint" pointcut="invoking(Joinpoint, 'SmsService',
      'SmsServicePT', Operation)">
2   <bpel:sequence>
3     <bpel:assign>
4       <bpel:copy>
5         <bpel:from>
6           <bpel:literal>Invoking operation $Operation...</bpel:literal>
7         </bpel:from>
8         <bpel:to variable="logMessageRequest" part="payload" />
9       </bpel:copy>
10    </bpel:assign>
11    <bpel:invoke partnerLink="LoggingService" portType="LoggingServicePT"
          operation="logMessage" inputVariable="logMessageRequest" />
12  </bpel:sequence>
13 </before>
```

Listing 3.5: An advice that logs the start of all invocations of the SmsService web service

BPEL code. For *before*, *after*, and *around* advices, this is a BPEL activity. For *in* advices, other BPEL elements can be used as well, such as a <faultHandlers> element. In an *around* advice, the <proceed> activity can be used to invoke the original behavior of the joinpoint. In all cases, the pointcut's arguments are exposed to the advice: these can be accessed in the advice by prefixing their name with the '$' character.

Listing 3.5 shows an example of a *before* advice that logs the start of all invocations of the SmsService web service (cf. line 1). The behavior that is added is a sequence of two activities: first, a log message containing the name of the operation to be invoked is created (cf. lines 3–10), and second, the log message is sent to the LoggingService web service (cf. line 11).

Listing 3.6 shows an example of an *around* advice that logs the start and end of all invocations of the SmsService web service (cf. line 1). The behavior that is added is a sequence of five activities: (1) a first log message containing the name of the operation to be invoked is created (cf. lines 3–10); (2) the first log message is sent to the LoggingService web service (cf. line 11); (3) the <proceed> activity indicates that the joinpoint's original behavior, i.e., the invocation of the SmsService, should be executed (cf. line 12); (4) a second log message containing the name of the operation that has been invoked is created (cf. lines 13–20); (5) the second log message is sent to the LoggingService web service (cf. line 21).

Listing 3.7 shows an example of an *in* advice that adds a fault handler to the CreateCall scope (cf. line 1). The fault handler catches all faults (cf. line 3) and handles these using a sequence of two activities: first, a log message containing a brief error message is created (cf. lines 4–11), and second, the log message is sent to the LoggingService web service (cf. line 12).

3.3.3 Aspect Module Model

An aspect represents a single crosscutting concern. As such, aspects can contain several *before*, *after*, *around*, and *in* advices. Listing 3.8 shows an example aspect that logs

```
1 <around joinpoint="Joinpoint" pointcut="invoking(Joinpoint, 'SmsService',
      'SmsServicePT', Operation)">
2   <bpel:sequence>
3     <bpel:assign>
4       <bpel:copy>
5         <bpel:from>
6           <bpel:literal>Invoking operation $Operation...</bpel:literal>
7         </bpel:from>
8         <bpel:to variable="logMessageRequest" part="payload" />
9       </bpel:copy>
10    </bpel:assign>
11    <bpel:invoke partnerLink="LoggingService" portType="LoggingServicePT"
          operation="logMessage" inputVariable="logMessageRequest" />
12    <proceed />
13    <bpel:assign>
14      <bpel:copy>
15        <bpel:from>
16          <bpel:literal>Invoked operation $Operation</bpel:literal>
17        </bpel:from>
18        <bpel:to variable="logMessageRequest" part="payload" />
19      </bpel:copy>
20    </bpel:assign>
21    <bpel:invoke partnerLink="LoggingService" portType="LoggingServicePT"
          operation="logMessage" inputVariable="logMessageRequest" />
22  </bpel:sequence>
23 </around>
```

Listing 3.6: An advice that logs the start and end of all invocations of the `SmsService` web service

```
1 <in joinpoint="Joinpoint" pointcut="executingScope(Joinpoint, 'CreateCall')">
2   <bpel:faultHandlers>
3     <bpel:catchAll>
4       <bpel:assign>
5         <bpel:copy>
6           <bpel:from>
7             <bpel:literal>A fault has been caught!</bpel:literal>
8           </bpel:from>
9           <bpel:to variable="logMessageRequest" part="payload" />
10        </bpel:copy>
11      </bpel:assign>
12      <bpel:invoke partnerLink="LoggingService" portType="LoggingServicePT"
            operation="logMessage" inputVariable="logMessageRequest" />
13    </bpel:catchAll>
14  </bpel:faultHandlers>
15 </in>
```

Listing 3.7: An advice that adds a fault handler to the `CreateCall` scope

```
1 <aspect name="logSmsServiceInvocations" xmlns:log="logging.example.com">
2   <using>
3     <bpel:partnerLink name="LoggingService"
          partnerLinkType="log:LoggingServicePLT" />
4     <bpel:variable name="logMessageRequest" messageType="log:logMessageRequest" />
5   </using>
6   <pointcut name="smsServiceInvocations(Joinpoint, Operation)"
        pointcut="invoking(Joinpoint, 'SmsService', 'SmsServicePT', Operation)" />
7   <advice name="logMessage(Message)">
8     <bpel:sequence>
9       <bpel:assign>
10        <bpel:copy>
11          <bpel:from>
12            <bpel:literal>$Message</bpel:literal>
13          </bpel:from>
14          <bpel:to variable="logMessageRequest" part="payload" />
15        </bpel:copy>
16      </bpel:assign>
17      <bpel:invoke partnerLink="LoggingService" portType="log:LoggingServicePT"
            operation="logMessage" inputVariable="logMessageRequest" />
18    </bpel:sequence>
19  </advice>
20  <before joinpoint="Joinpoint" pointcut="smsServiceInvocations(Joinpoint,
        Operation)">
21    <advice name="logMessage('Invoking operation $Operation...')" />
22  </before>
23  <after joinpoint="Joinpoint" pointcut="smsServiceInvocations(Joinpoint,
        Operation)">
24    <advice name="logMessage('Invoked operation $Operation')" />
25  </after>
26 </aspect>
```

Listing 3.8: An aspect that logs the start and end of all invocations of the `SmsService` web service

the start and end of all invocations of the `SmsService` web service using a before and after advice, respectively.[1] The main sections of an aspect are the `<using>` declaration (lines 2–5), the pointcut (line 6) and advice definitions (lines 7–19), and the actual advices (lines 20–25). In order to allow a rudimentary form of reuse of pointcuts and advices, aspects can include other aspect files using one or more `<include>` declarations (cf. our case study in Section 3.4).

Adding new behavior to a process usually requires extending the information defined at the process level in addition to adding the behavior itself. For example, adding a new invocation to a process usually requires adding a partner link that specifies the interface of the service to be invoked, and a new variable that will contain the message that should be sent to that service. The `<using>` declaration (lines 2–5) allows the definition of such additional process level information. These may include partner links, variables, compensation handlers, fault handlers, termination handlers, and event handlers.

[1] Alternatively, the aspect could use a single *around* advice, such as the advice in Listing 3.6.

```
1 <deployment>
2   <!-- The following aspects need to be applied to the specified processes: -->
3   <aspect name="logSmsServiceInvocations" process="ConferenceCall"
        id="ConferenceCallLogging" />
4   <aspect name="performFixedFeeBilling" process="ConferenceCall"
        id="ConferenceCallFixedFeeBilling" />
5   <!-- The following precedence rules are valid for the specified process, or for
        all processes if no process is specified: -->
6   <precedence process="ConferenceCall" />
7     <aspect id="ConferenceCallLogging" advice="before" />
8     <aspect id="ConferenceCallFixedFeeBilling" advice="before" />
9     <aspect id="ConferenceCallFixedFeeBilling" advice="after" />
10    <aspect id="ConferenceCallLogging" advice="after" />
11  </precedence>
12 </deployment>
```

Listing 3.9: An aspect deployment specification

Pointcuts can be reused by giving them a name and specifying their arguments (line 6), which can either be further constrained when reusing the expression, or be referred to from inside an advice that reuses the pointcut. Defining a pointcut like this gives rise to a high-level pointcut predicate that can later be used in other pointcuts.

The extra behavior that is to be inserted using *before*, *after*, *around*, and *in* advices can be reused as well (lines 7–19). The advice behavior is given a name and can be parametrized. These parameters can be referred to from inside the advice code using their name prefixed with the '$' character. The named advice behavior can be invoked from within advice code using the `<advice>` element (lines 21 and 24).

3.3.4 Aspect Instantiation and Composition Models

An aspect represents a single crosscutting concern. Because more than one crosscutting concern can be applicable to the same process, the instantiation and composition of a process's aspects should be performed in a way that prevents undesirable interactions between aspects. Therefore, PADUS introduces an explicit deployment construct that specifies how aspects should be applied to a base process (or base processes).

Listing 3.9 illustrates aspect deployment in PADUS. A PADUS aspect deployment consists of two main parts: aspect instantiation (lines 3–4) and aspect composition (lines 6–11). Aspect instantiation is responsible for instantiating aspects and applying them to the appropriate processes. Processes are referenced using their name. The PADUS weaver (cf. Section 3.5) will determine which aspects need to be applied to which processes based on this part of the aspect deployment.

The second part of an aspect deployment, namely the aspect composition, is responsible for specifying the precedence of aspects in case multiple aspects apply to the same joinpoint. In case no precedence is specified, the advice is executed in the order in which their corresponding aspects are specified. A precedence declaration overrides this default and is able to specify precedence on a per-advice-type basis. Aspect precedence for a *before* advice can thus be different than precedence for an *after* advice, as is the

```
1 <aspect name="performBilling" xmlns:bill="billing.example.com">
2   <using>
3     <bpel:partnerLink name="BillingService"
          partnerLinkType="bill:BillingServicePLT" />
4     <bpel:variable name="billRequest" messageType="bill:billRequest" />
5   </using>
6   <pointcut name="confCallStarts(Joinpoint)" pointcut="invoking(Joinpoint,
        'ConfCallService', 'ConfCallServicePT', 'createConfCall')" />
7   <pointcut name="confCallEnds(Joinpoint)" pointcut="invoking(Joinpoint,
        'ConfCallService', 'ConfCallServicePT', 'closeConfCall')" />
8   <advice name="bill">
9     <bpel:invoke partnerLink="BillingService" portType="bill:BillingServicePT"
          operation="bill" inputVariable="billRequest" />
10  </advice>
11 </aspect>
```

Listing 3.10: Aspect defining generic billing concepts

case in Listing 3.9. The precedence may also vary over several deployments of the same aspect type, as it is bound to the aspect instance and not to the aspect type. Furthermore, the precedence specification can be limited to specific processes, allowing a custom precedence specification for each process or group of processes if necessary. This aspect precedence scheme was influenced by the connector construct in JAsCo (cf. Section 2.2.2). PADUS's explicit deployment construct is an improvement on existing AOP approaches for workflows, where aspect deployment is implicit.

3.4 Case Study: The Billing Concern

In this section, we show how PADUS can be used to add the billing concern to a multi-party conference call process. Two types of billing schemes are supported: a *fixed fee billing scheme* where the end user should pay a fixed price at the end of the conference call, and a *duration billing scheme* where the price is determined by the duration of the conference call. Three aspects are used to represent these two billing schemes:

1. A *generic billing aspect* (cf. Listing 3.10) defines concepts common to both billing schemes: the billing service partner link and message definitions (lines 2–5), the pointcuts representing the start and end of a conference call (lines 6 and 7), and an advice for invoking the billing service (lines 8–10).

2. The *fixed fee billing aspect* (cf. Listing 3.11) defines one advice, which invokes the billing service with a fixed price of 1.50 EUR at the end of the conference call.

3. The *duration billing aspect* (cf. Listing 3.12) defines two advices: the first advice (lines 6–13) stores the start time of the conference call in a new variable (line 4), while the second advice (lines 14–29) uses this time to calculate the price of the conference call based on its duration and then invokes the billing service.

```
<aspect name="performFixedFeeBilling">
  <include name="performBilling" />
  <after joinpoint="Joinpoint" pointcut="confCallEnds(Joinpoint)">
    <bpel:sequence>
      <bpel:assign>
        <bpel:copy>
          <bpel:from>
            <bpel:literal>
              <price currency="EUR" xmlns="billing.example.com/xsd">
                1.50
              </price>
            </bpel:literal>
          </bpel:from>
          <bpel:to variable="billRequest" part="payload" />
        </bpel:copy>
      </bpel:assign>
      <advice name="bill" />
    </bpel:sequence>
  </after>
</aspect>
```

Listing 3.11: Aspect implementing a fixed fee billing scheme

The logic needed for adding billing to the conference call process is now cleanly modularized in these aspects. Without an aspect mechanism, the BPEL code that is specified in the aspects' advice would need to be inserted at each of the joinpoints manually, resulting in significant code duplication across the process. Thus, the aspect mechanism helps to keep the complexity of the process under control. Any of the two concrete billing aspects can now be combined with the conference call process, or any other process, which facilitates reuse. The billing scheme can now be modified more easily, as well, because all billing logic is encapsulated in just a few aspects.

The aspect deployment specification that was already provided in Listing 3.9 specifies how the above aspects should be instantiated and composed: both the `logSmsServiceInvocations` and `performFixedFeeBilling` aspects are applied to the `ConferenceCall` process, and the precedence of both aspects' before and after advices is specified.

3.5 Architecture and Implementation

3.5.1 Architecture

The implementation of PADUS employs a statically woven approach, which guarantees the absence of runtime overhead. This choice is guided by the application domain: because PADUS is used to describe real-time processes in a telecom Service Delivery Platform, performance is important. Another important advantage is that a statically woven approach does not require a dedicated execution platform, which could limit the compatibility of the implementation with existing tool chains.

```
<aspect name="performDurationBilling">
  <include name="performBilling" />
  <using>
    <bpel:variable name="startTime" type="func:time" />
  </using>
  <before joinpoint="Joinpoint" pointcut="confCallStarts(Joinpoint)">
    <bpel:assign>
      <bpel:copy>
        <bpel:from>func:getCurrentTime()</bpel:from>
        <bpel:to variable="startTime" />
      </bpel:copy>
    </bpel:assign>
  </before>
  <after joinpoint="Joinpoint" pointcut="confCallEnds(Joinpoint)">
    <bpel:sequence>
      <bpel:assign>
        <bpel:copy>
          <bpel:from>
            func:calculatePrice(
              bpel:getVariableProperty("startTime", "func:time"),
              "EUR",
              "0.40")
          </bpel:from>
          <bpel:to variable="billRequest" part="payload" />
        </bpel:copy>
      </bpel:assign>
      <advice name="bill" />
    </bpel:sequence>
  </after>
</aspect>
```

Listing 3.12: Aspect implementing a duration billing scheme

Figure 3.1 illustrates the architecture of the PADUS weaver. The weaver statically weaves a number of aspects into a base BPEL process according to an aspect deployment specification. The result is a woven BPEL process that can be deployed on any standard BPEL engine, such as ACTIVEBPEL (Active Endpoints, 2006) or ODE (Apache Software Foundation, 2009). In the WIT-CASE project, the BPEL engine was a part of a larger Service Delivery Platform, in which the engine was linked to existing telecom services using an Enterprise Service Bus. A Service Creation Environment (SCE) has been developed, which allows visual configuration of telecom service compositions based on BPEL and PADUS. This SCE is discussed in detail in Section 3.6.

3.5.2 Weaver Implementation

The PADUS weaver can be downloaded from the PADUS website (Joncheere et al., 2009). Because our pointcut language is based on logic concepts, we have opted to implement our weaver using PROLOG. The weaver's input are the path to an input directory (containing a BPEL4WS process and any number of PADUS aspects), the name of a PADUS deployment file, and the path to an output directory. The weaver will start by parsing the

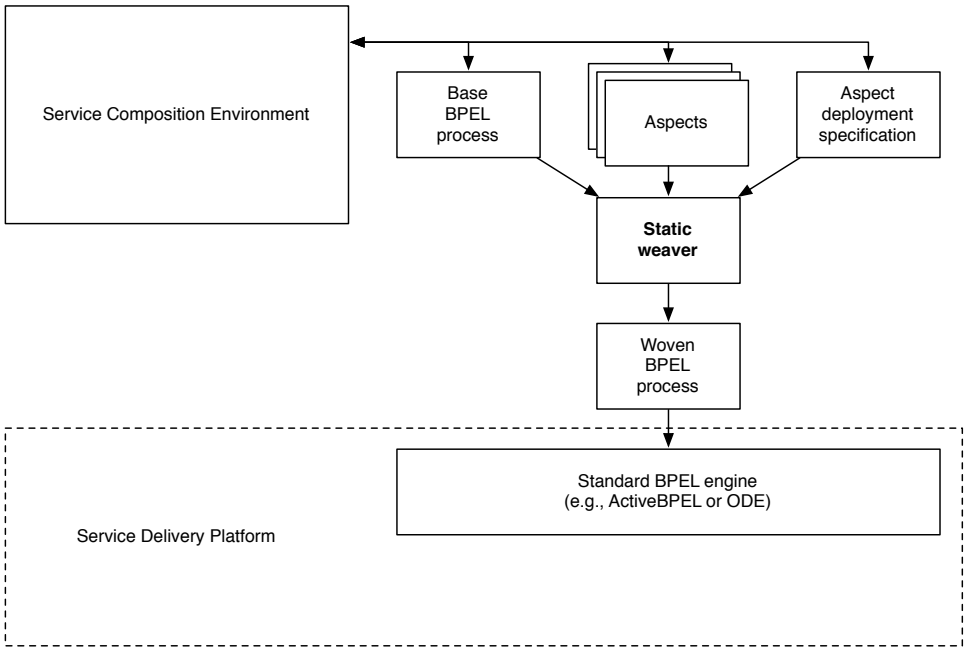


Figure 3.1: PADUS weaver architecture

deployment file. The referenced BPEL4WS process and each of the referenced PADUS aspects are parsed using the SWI-PROLOG XML parser and are thus asserted as PROLOG facts. The weaver will then iterate over each of the PADUS aspects' advices in the order specified by the deployment file, and will change the BPEL4WS process accordingly.

For each advice, the specified logic query is evaluated in order to determine the join-points where the advice's behavior should be added. It is at this point that the benefits of using PROLOG for implementing our weaver are most evident, as it greatly facilitates implementing our high-level pointcut language.[2] After the query has been evaluated, the behavior is added to the BPEL4WS process. When all the advices have been woven, the modified BPEL4WS process is translated from PROLOG facts back to BPEL4WS XML code, and the weaver terminates.

3.6 The Service Creation Environment

3.6.1 Overview

In the context of the WIT-CASE project, we have developed a Service Creation Environment (SCE) that allows user-friendly composition of services (Braem et al., 2006a,b). The

[2] The weaver could be implemented just as well in a more traditional language, but this would likely require more effort to implement the pointcut language.

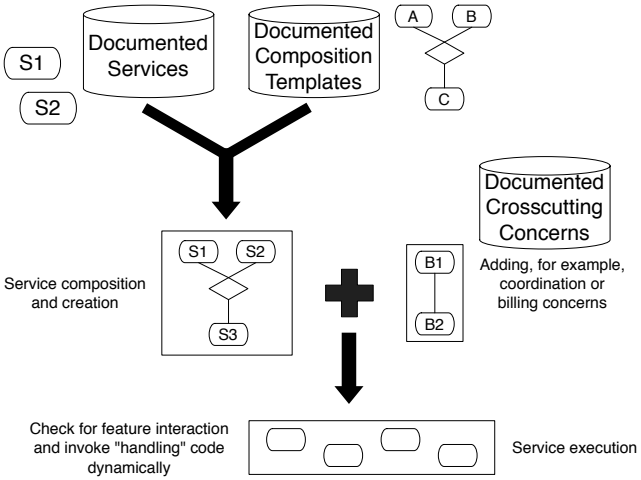


Figure 3.2: Service Creation Environment architecture

architecture of the SCE is illustrated by Figure 3.2. The SCE contains three repositories:

1. A set of *documented services*, which are the basic building blocks of the SCE. Each service's interface is documented by a WSDL description. Optionally, a service's WSDL description may specify a number of basic quality of service properties, or a service's external protocol may be documented by a BPEL4WS process.

2. A set of *documented composition templates*, which allow reusing predefined patterns for composing services. Composition templates are represented by abstract BPEL4WS processes, and can be generated from existing BPEL4WS processes or BPMN diagrams. Composition templates are not yet bound to concrete services, but instead contain *service placeholders*, which may specify quality of service constraints.

3. A set of *documented crosscutting concerns*, which allow applying predefined crosscutting behavior to service compositions. Crosscutting concerns are represented by PADUS aspects. Similar to composition templates, crosscutting concerns are not yet bound to concrete services, and may thus contain placeholders.

Figure 3.3 is a screenshot of the SCE. At the center of the screen, an editor view contains a large canvas and a smaller palette. The canvas shows a graphical view of the service composition that is being created. The palette lists the available services, composition templates and crosscutting concerns, and allows dragging these to the canvas. Composition templates can be bound to concrete services by dragging services onto their placeholders. Crosscutting concerns can be connected to composition templates in order to apply crosscutting behavior such as billing. By double-clicking on a service or composition template, the configured editor for that service or composition template

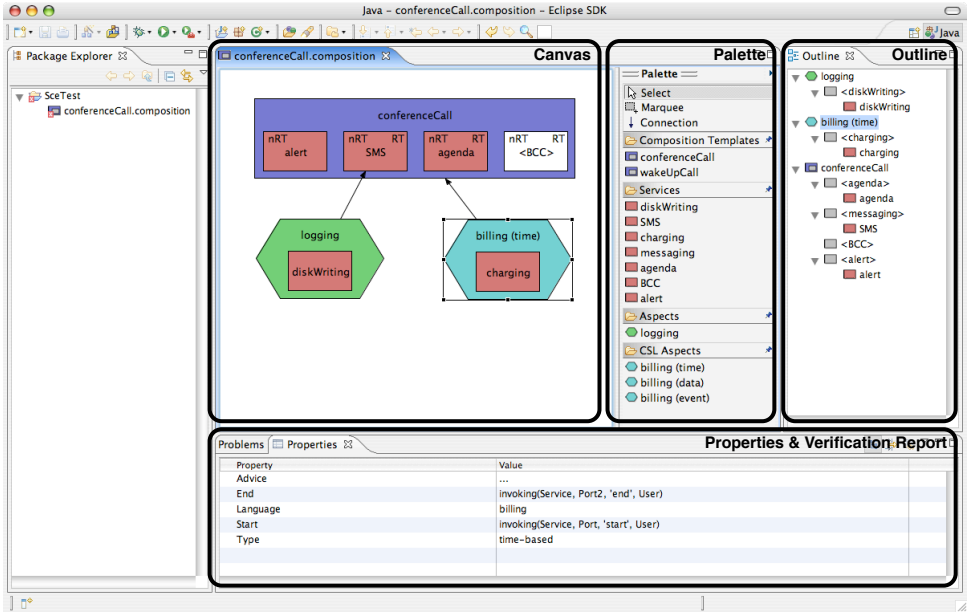


Figure 3.3: Service Creation Environment screenshot

is launched. By default, a BPMN editor is used, but other process modeling notations or languages can be used to visualize the internal representation of a service or composition template, as well. The outline view (at the right of the screen) shows a tree-based overview of the composition, and the properties view (at the bottom of the screen) shows the properties of the element that is currently selected in the editor view or in the outline view.

Based on the documentation of the composed elements, the SCE can guide the user in creating valid service compositions. In Section 3.6.2, we explain how this is accomplished.

3.6.2 Guiding the Service Composition Process

3.6.2.1 Protocol Verification

An important requirement of the SCE is that it supports and guides users in creating valid compositions. The SCE accomplishes this by verifying whether compositions are valid while they are created: when a service is dragged onto a placeholder, the SCE checks whether the service's protocol is compatible with the composition template's protocol. In case the service turns out to be incompatible, a report is generated that provides mismatch feedback to the user. Compatibility checking based on protocols rather than plain APIs is possible because every service is explicitly documented with a protocol specification expressed in BPEL.

In literature, a wealth of research exists on the topic of protocol verification (Campbell and Habermann, 1974; van den Bos and Laffra, 1991; Luckham et al., 1995; Yellin and Strom, 1997; Reussner, 2003). Our verification engine is based on the PacoSuite approach (Wydaeghe, 2001), which introduces algorithms based on automata theory to perform protocol verification. In order to provide protocol verification in the SCE, the BPEL specifications of each service, aspect and composition template are translated into deterministic finite automata (DFA). By applying the algorithms introduced by the PacoSuite approach, the SCE can decide whether the service's protocol is compatible with the composition template's protocol. More specifically, the PacoSuite algorithm consists of constructing an automaton that covers the complete composition by taking a specialized intersection of the composition template's automaton with each service's automaton. The composition is invalid when the resulting automaton is empty (i.e., no path from a start to stop state). In that case a report is generated that pinpoints which operations of which service does not follow the composition template's protocol. In case more detail is required, the user can also view the generated automaton. Because the algorithm is based on taking intersections of automatons, the resulting performance is in the worst case exponential with respect to the size of the input automata. Notice that the algorithm is only executed at composition time and thus does not interfere with the running application.

It is possible that an aspect adapts the external protocol of an existing service (e.g., by adding an invocation) so that it becomes incompatible with the composition template's protocol. Our tool therefore employs the PADUS weaver both on the composition template and on the services' BPEL protocol specification before translating them to DFAs. As such, the effect of the aspect on the external protocol of the composition template and services is visible and can be taken into account by the verification engine.

3.6.2.2 Guideline Verification

In the context of the WIT-CASE project, the partners have identified a non-exhaustive list of conditions that can apply to service compositions in order to ensure efficient execution. The SCE enables the implementation of guidelines that statically check these conditions and detect bad smells in service compositions. These guidelines are optional and can be enabled and disabled by the service designer in the SCE. We list some of these guidelines here:

- **Quality-of-service.** In a composition template, execution time constraints may be specified on placeholders for services, which limit the services that can be used to those that have an execution time bounded within these constraints.

- **Short-lived real-time processes.** This guideline is closely related to the quality-of-service guideline, but takes the execution time of the complete real-time part of the composition (instead of operations on single services) into account. The minimal execution time of the real-time process can be computed from the modeled service composition, and the SCE generates a warning if this execution time exceeds a certain predefined amount of time.

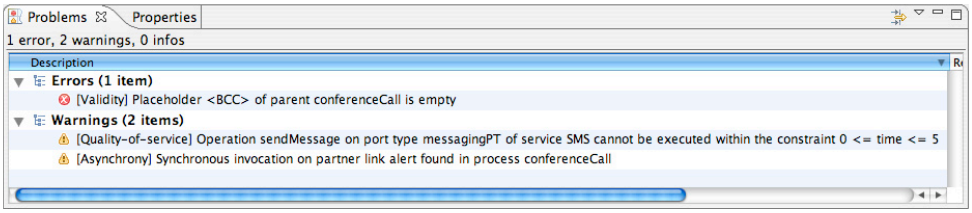


Figure 3.4: SCE guideline verification report

- **Concurrency.** Handling tasks concurrently improves efficiency because independent tasks do not have to wait for others to finish. This guideline verifies if activities are situated in concurrent branches.

- **Asynchrony.** The asynchrony guideline verifies that the invocations used in the composition are always asynchronous.

To illustrate the quality-of-service guideline we go back to the running example. The composition template requires a messaging service to notify the participants of the conference call of any errors. Imagine that the composition designer has two different services available that handle messaging: one works by sending a text message to the participants' phones, the other by sending an email to the participants. In the documentation for these services, it is declared that sending an SMS message takes 10 time units, while sending an email takes 5 time units. The composition templates states that using the messaging service can take a maximum of 5 time units, so only the email messaging service is accepted by the guideline.

Figure 3.4 shows a screenshot of the output of the guideline verification in the SCE. Two warnings are shown. The first one reports the violation of the quality-of-service by the concrete SMS service as explained above. The second warning indicates that the asynchrony guideline is violated because a synchronous invocation is found in the composition specification of the conference call composition. The error shown in the screenshot in Figure 3.4 specifies that the composition is not complete because the "BCC" placeholder is not yet filled in.

3.6.2.3 Code Generation and Deployment

When the composition is complete and verified, the user may choose to generate the resulting composition and deploy it on a BPEL engine. This will start the code generation process, which will bind the unbound partner links in the composition templates. An aspect deployment is automatically generated for the aspects contained in the composition. The PADUS weaver is then employed to weave the aspects into the resulting BPEL processes based on the aspect deployment specification. A resulting composition can be imported back into the library as a new service. The generated BPEL process then serves as documentation for the new service. Apart from specifying a name for the new service, this process is also automated.

3.7 Summary

In this chapter, we present the PADUS language, an aspect-oriented extension to BPEL that constitutes our first approach for modularization of concerns in workflows. Inspired by the context of the WIT-CASE project, which addressed the orchestration of web services in a telecom Service Delivery Platform using BPEL, the approach promotes separation of concerns in BPEL processes by allowing crosscutting concerns to be specified in separate aspects. It addresses the requirements listed in Section 3.2 as follows:

1. We provide a rich joinpoint model that consists of all BPEL activities.
2. We employ a high-level, logic pointcut language that makes pointcuts less dependent on the concrete XML document structure of BPEL processes. The pointcuts are thus less fragile with respect to evolution of the base processes. The pointcut language allows reuse of user-defined pointcut predicates.
3. We support the three basic advice types that were identified in general aspect-oriented research, i.e., *before*, *after*, and *around*.
4. We introduce the concept of an *in* advice to add new behavior to existing workflow elements, and thus address some of the specifics of the workflow paradigm.
5. We use BPEL as the advice language.
6. We provide an explicit deployment construct that allows instantiating aspects and applying them to BPEL processes. Aspect composition is tackled by a precedence specification that may vary across different BPEL processes, aspect instances, and advice types.
7. We provide an efficient implementation strategy, i.e., static weaving of BPEL processes and PADUS aspects.
8. We promote compatibility with the existing tool chain, as we do not impose a dedicated BPEL engine.

Table 3.6 compares PADUS to the state of the art in AOP for BPEL as it was discussed in Section 3.2. It should be clear that PADUS addresses our requirements significantly better than AO4BPEL and the approach by Courbis and Finkelstein.

Although the above contributions imply a significant improvement over the state of the art in aspect-oriented programming for workflows, we believe that by going beyond the specific context of the WIT-CASE project and adding a number of features compared to our first approach, we can obtain an approach that facilitates separation of concerns in workflows even more. This second approach is developed in the following chapters.

Requirement	AO4BPEL	Courbis and Finkelstein	PADUS
A rich joinpoint model	−	+	+
A high-level pointcut language	−	−	+
Basic advice types	+	±	+
Workflow-specific advice types	−	−	±
BPEL as the advice language	+	+	+
An explicit deployment construct	−	−	+
An efficient implementation strategy	±	±	+
Compatibility with the existing tool chain	−	−	+

Table 3.6: Evaluation of PADUS

Chapter 4

Uniform Modularization of Workflow Concerns using UNIFY

This chapter presents a more ambitious approach for modularization of concerns in workflows than the one presented in the previous chapter. The approach, which is called UNIFY*, has a wider scope than* PADUS *in that it is aimed at modularizing* all *workflow concerns (i.e., not only crosscutting ones), provides advice types that recognize the specific characteristics of the workflow paradigm, and is applicable to multiple workflow languages. In this chapter, we describe our motivation and requirements, provide an overview of the approach, and introduce the meta-models that lie at the heart of* UNIFY*. In Chapters 5–7, we discuss a number of other essential characteristics of the approach, i.e., its execution semantics, domain-specific layer, and implementation, in more detail.*

4.1 Motivation and Requirements

The specific context in which PADUS was developed, i.e., the WIT-CASE project, has guided the design of the approach. When evaluating the scope of the approach, the following limitations can be observed:

- PADUS only targets modularization of *crosscutting* concerns.
- PADUS only provides one workflow-specific advice type, i.e., the *in* advice type.
- PADUS only targets BPEL, which — though the most popular — is not the only workflow language available.

By addressing these limitations, we can obtain an approach that is more widely applicable and that tackles the problem of separation of concerns in workflows in a more effective way. Therefore, this chapter will introduce our second approach for modularization of concerns in workflows, which is significantly wider in scope than PADUS.

Consider the order handling workflow in Figure 4.1, which we have already used in Chapter 2. The workflow starts at the start event at the top of the figure. It first performs the *Login* and *SelectBooks* activities in parallel. The workflow then proceeds with the *SpecifyOptions* activity, after which the control flow is split again. A first branch contains the *Pay* and *SendInvoice* activities, while a second branch contains the *ProcessOrder* and *Ship* activities. The *VerifyBankAccount* activity synchronizes both branches: the *Ship* activity cannot be executed before the *Pay* and *VerifyBankAccount* activities have been executed. The last activity to be executed is the *ProcessReturns* activity, after which the workflow ends at the end event. Note that only the contents of the *SelectBooks*, *Pay*, and *Ship* activities are shown, whereas the contents of other activities are omitted in the interest of clarity. In addition to its specification in BPMN, we have implemented the order handling workflow in WS-BPEL.[1] In this implementation, the workflow specification as it is shown in Figure 4.1 is augmented with a data perspective and activities related to user interaction.

Like any realistic software application, the workflow in Figure 4.1 consists of several *concerns* — parts that are relevant to a particular concept, goal, or purpose — which are connected in order to achieve the workflow's desired behavior. The main concern is obviously *order handling*. This concern has already been hierarchically decomposed into sub-concerns — such as *book selection*, *payment*, and *shipping* — using the *composite activity* construct. Other concerns are *preference saving*, *reporting*, and *bank account verification*, which occur at various places across the workflow. The general software engineering notion of *separation of concerns* (Dijkstra, 1982) refers to the ability to identify, encapsulate, and manipulate such concerns in isolation of each other. Separation of concerns is traditionally accomplished by decomposing software into modules, which is associated with benefits regarding development time, product flexibility, and comprehensibility (Parnas, 1972). However, many current workflow languages do not allow effectively decomposing workflows into different modules. For example, a workflow expressed using BPEL is a single, monolithic XML file that cannot be straightforwardly divided into sub-workflows. This lack of modularization mechanisms makes it hard to add, maintain, remove, or reuse concerns. In order to improve separation of concerns in workflows, workflow languages should allow concerns to be specified in isolation of each other.

However, allowing concerns to be specified in isolation of each other is not sufficient: in order to obtain the desired workflow behavior, workflow languages should also provide a means of specifying how workflow concerns are connected to each other. In existing workflow languages, the only kind of connection that is supported is typically the classic sub-workflow pattern: a main workflow explicitly specifies that a sub-workflow should be executed. The choice of which sub-workflow is to be executed is made at design time, and it is hard to make a different choice afterwards. A mechanism that reduces the coupling between main workflow and sub-workflow is therefore desirable. In

[1] The size of the resulting XML document prevents us from providing it as an appendix to this dissertation. The entire WS-BPEL specification, together with related documents such as WSDL descriptions of the workflow and any referenced web services, as well as the decomposition of the WS-BPEL specification into separate concerns, can be downloaded from the UNIFY website at `http://www.unify-framework.org/OrderBooks.tgz`.

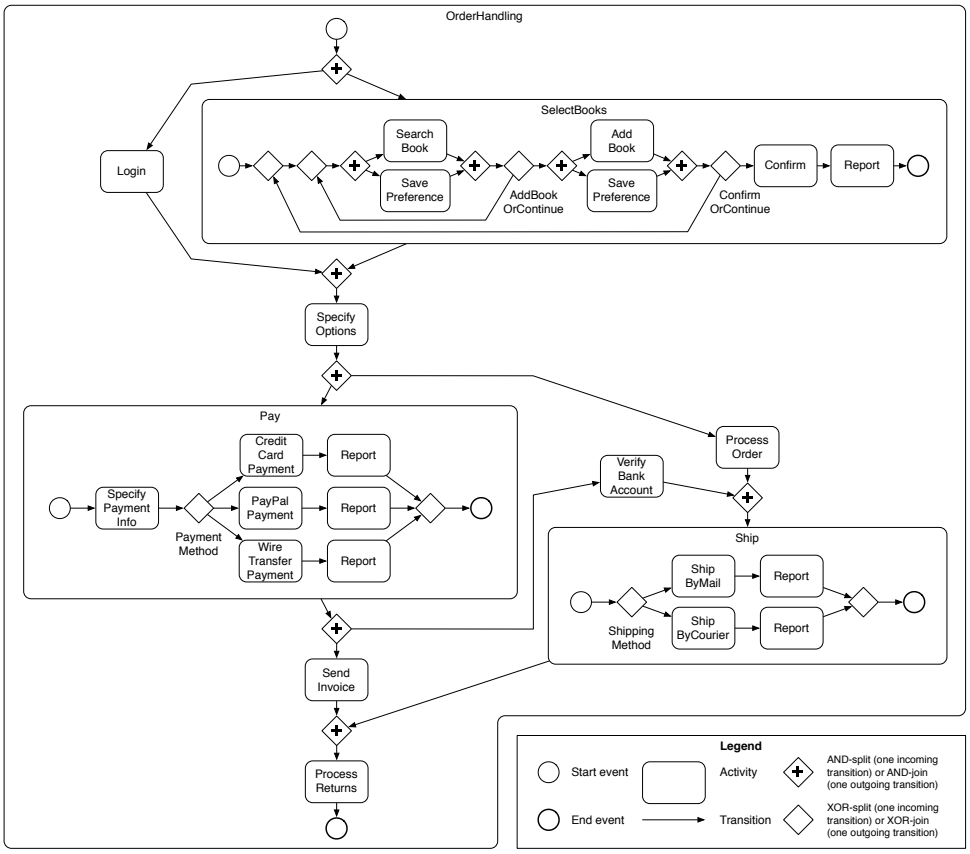


Figure 4.1: Example order handling workflow, expressed using BPMN

the workflow in Figure 4.1, one could for example vary the behavior of the workflow by deploying a different *Pay* sub-workflow in different situations.

A second kind of connection between concerns is useful when concerns *crosscut* a workflow: some concerns cannot be modularized cleanly using the sub-workflow decomposition mechanism, because they are applicable at multiple locations in the workflow. The reporting concern, for example, is present at several locations in the workflow in Figure 4.1. The sub-workflow construct does not solve this problem, since sub-workflows are called explicitly from within the main workflow. This makes it hard to add, maintain, remove or reuse such crosscutting concerns. This problem has been observed in general aspect-oriented research (Kiczales et al., 1997). Aspect-oriented extensions to BPEL, such as AO4BPEL, the approach by Courbis and Finkelstein, and PADUS, allow specifying crosscutting concerns in separate *aspects*. An aspect allows specifying that a certain workflow fragment, called an *advice*, should be executed *before*, *after*, or *around* a specified set of activities in the base workflow. In the workflow in Figure 4.1, one could for

example specify that the *Report* activity needs to be performed after the *Confirm* activity and after each of the three *Payment* and two *Ship* activities, without explicitly invoking the *Report* activity at each of those places. However, these aspect-oriented extensions use a new language construct for specifying crosscutting concerns, i.e., aspects. This means that concerns which are specified using the aspect construct can only be reused as an aspect, and not as a sub-workflow. On the other hand, concerns which are specified using the sub-workflow construct can only be reused as a sub-workflow, and not as an aspect. Thus, we identify the following requirement:

Requirement 1. *A uniform modularization mechanism that allows specifying both regular and crosscutting concerns using the same language construct.*

Moreover, existing aspect-oriented extensions only support the basic concern connection patterns (*before*, *after*, or *around*) that were identified in general aspect-oriented research, and do not sufficiently consider the specific characteristics of the workflow paradigm. They lack support for other patterns such as parallelism and choice. For example, the before, after, or around patterns do not provide an elegant way of specifying that the *SavePreference* activity should be performed *in parallel with* the *SearchBook* and *AddBook* activities. Furthermore, it is completely impossible to specify more advanced connections between concerns, e.g., specifying that the *VerifyBankAccount* activity should be executed after the *Pay* activity has been executed and before the *Ship* activity is executed, which would thus synchronize the two parallel branches by introducing a new AND-split and -join in the order handling workflow. Thus, we identify the following requirements:

Requirement 2. *A coherent collection of workflow-specific concern connection patterns.*

Requirement 3. *A means of connecting workflow concerns according to the above concern connection patterns.*

Because the correct execution of workflows is of vital importance to an organization, a long tradition of formal verification of workflows exists. For example, the execution semantics of WS-BPEL has been formalized using Petri nets (Lohmann, 2007), and YAWL has been developed by augmenting high-level Petri nets with additional constructs (thus obtaining *extended workflow nets*; cf. van der Aalst and ter Hofstede, 2005). Because new modularization mechanisms have a significant impact on the semantics of their base language, it is important that these modularization mechanisms fit into this existing formal tradition. Thus, we identify the following requirement:

Requirement 4. *An execution semantics that is compatible with existing research on execution semantics of workflow languages.*

The abstractions offered by existing modularization approaches for workflows typically remain at the same level as the base workflow: concerns are implemented using the constructs of the base workflow language, which may not be ideally suited for expressing the concern in question. Although existing aspect-oriented extensions improve separation of concerns, they introduce additional complexity in the implementation of

a workflow. This complexity must be bridged in order to communicate the implementation of a workflow to the domain experts who identified the process that is automated by the workflow. For example, if we were to add some kind of access control concern to the workflow in Figure 4.1 that specifies that only premium customers are allowed to specify options using the *SpecifyOptions* activity, we would need to introduce a new XOR-split and XOR-join around the activity, with conditions that query the data perspective of the workflow in order to find out whether the currently logged in user is a premium customer or not. This requires detailed knowledge of how users are represented in the workflow's implementation. Inspired by the benefits of domain-specific languages in software engineering (van Deursen et al., 2000), we believe that a means of expressing workflow concerns using abstractions that are close to the concerns' domains can facilitate expressing workflow concerns, and can improve communication with domain experts. Thus, we identify the following requirement:

Requirement 5. *A means of expressing workflow concerns using abstractions that are close to the concerns' domains.*

An additional disadvantage of existing aspect-oriented extensions to workflow languages is that they are all targeted at BPEL, and cannot be applied easily to other workflow languages. Each of these extensions also favors a specific implementation technique; for example, AO4BPEL and the approach by Courbis and Finkelstein can only be executed using a dedicated BPEL4WS engine. Although this has the advantage that it facilitates adding dynamic features to the modularization mechanism, it precludes compatibility with existing tool chains. We aim for our approach to be applicable to several concrete workflow languages, and to be independent of a dedicated workflow engine. Thus, we identify the following requirements:

Requirement 6. *A means of applying the modularization mechanism to several concrete workflow languages.*

Requirement 7. *An implementation that is independent of a dedicated workflow engine.*

This concludes the identification of our requirements. In the following section, we introduce the approach taken to fulfill these requirements.

4.2 Approach

Our approach, which is called UNIFY, addresses each of the requirements we identified in the previous section, and which are summarized in Table 4.1. At the heart of UNIFY lies a base language meta-model that allows specifying both regular and crosscutting concerns using the same construct (Requirement 1). We identify a coherent collection of patterns that describe the ways in which such uniform workflow concerns can be connected to each other (Requirement 2). We propose a connector mechanism that allows connecting workflow concerns according to these patterns (Requirement 3). Both our base language and our connection mechanism have an explicit execution semantics based on Petri nets (Requirement 4). On top of the base language and connector

Requirement	Cf.
1. A uniform modularization mechanism	Section 4.3
2. Workflow-specific concern connection patterns	Section 4.4
3. A means of connecting workflow concerns	Section 4.5
4. An explicit execution semantics	Chapter 5
5. Support for concern-specific abstractions	Chapter 6
6. Applicable to several concrete languages	Chapter 7
7. Independent of a dedicated workflow engine	Chapter 7

Table 4.1: Requirements for UNIFY

mechanism, we build a concern-specific layer that allows specifying concerns using abstractions that map closely to the domain of the concern at hand (Requirement 5). The base language meta-model is instantiated towards several concrete workflow languages (Requirement 6), and our implementation is a pre-processor that is compatible with existing tool chains (Requirement 7).

In the following sections and chapters, we describe each of the above topics in more detail. The UNIFY base language is described in Section 4.3. In Section 4.4, we identify a coherent set of workflow-specific concern connection patterns, which are implemented by our connector mechanism that is described in Section 4.5. Our semantics is described in Chapter 5, our concern-specific layer is described in Chapter 6, and the instantiation and implementation of UNIFY are described in Chapter 7. In Table 4.1, we specify the mapping from each of our requirements to each of these sections and chapters.

4.3 Base Language

In this section, we describe the UNIFY base language, which defines how workflow concerns are implemented as independent modules. The base language's control flow perspective is introduced in Section 4.3.1, whereas the data perspective is introduced in Section 4.3.2.

4.3.1 Control Flow Perspective

UNIFY is designed to be applicable to a range of concrete workflow languages, as long as they conform to a number of basic assumptions. These assumptions are expressed in UNIFY's meta-model for implementing workflow concerns. We do not restrict ourselves to any particular concrete workflow language as long as it can be defined as an extension to this meta-model. The meta-model allows expressing *arbitrary workflows* (Kiepuszewski et al., 2000), i.e., workflows in which every split does not necessarily need to have a corresponding join, and is therefore also compatible with more restricted workflows such as *structured workflows* (Kiepuszewski et al., 2000).

Figure 4.2 provides the meta-model for the control flow perspective of the UNIFY base language. The meta-model is expressed using UML class diagrams (Object Management Group, 2007), with further well-formedness constraints specified in OCL (Ob-

ject Management Group, 2006). A workflow concern is modeled as a CompositeActivity. Each CompositeActivity has the following children:

1. A StartEvent, which represents the point where the CompositeActivity's execution starts.

2. An EndEvent, which represents the point where the CompositeActivity's execution ends.

3. Any number of Activities, which are the units of work that are performed by the CompositeActivity.

4. Any number of ControlNodes, which are used to route the CompositeActivity's control flow.

5. One or more Transitions, which connect the StartEvent, the EndEvent, the Activities and the ControlNodes to each other.

An Activity is either a CompositeActivity or an AtomicActivity. Nested CompositeActivities can be used to hierarchically decompose a concern, similar to the classic subworkflow decomposition pattern. Each Activity has a name that is unique among its siblings in the composition hierarchy, and has one ControlInputPort and one ControlOutputPort. A ControlInputPort represents the point where control enters an Activity, while a ControlOutputPort represents the point where control exits an Activity. Each ControlPort has a name that is unique among its siblings. Within a CompositeActivity, the StartEvent is used to specify where the CompositeActivity's execution should start when its ControlInputPort is triggered. The EndEvent is used to specify where the CompositeActivity's execution should finish, and will cause the CompositeActivity's ControlOutputPort to be triggered. Thus, a StartEvent only has a ControlOutputPort, and an EndEvent only has a ControlInputPort.

Transitions define how control flows through a CompositeActivity. This is done by connecting the ControlOutputPorts of the CompositeActivity's Nodes to ControlInputPorts. ControlNodes can be used to route the control flow, and are either AndSplits, XorSplits, AndJoins or XorJoins. A Split may have a corresponding Join. Together, Transitions and ControlNodes define a CompositeActivity's control flow perspective.

As is shown in Table 4.2, the meta-model supports the *basic control flow patterns* (Russell et al., 2006a), which are sufficient for expressing most workflows. We do not aim to support more advanced patterns such as *cancellation patterns* and *multiple instances patterns*, as our focus lies with the expressiveness of the modularization mechanism rather than with the expressiveness of the individual modules. Due to the generic nature of the UNIFY base language meta-model, the cores of most workflow languages are compatible with it. For example, we have extended the meta-model towards the cores of the WS-BPEL and BPMN workflow languages, which are the two languages most popular in current workflow research (cf. Chapter 7).

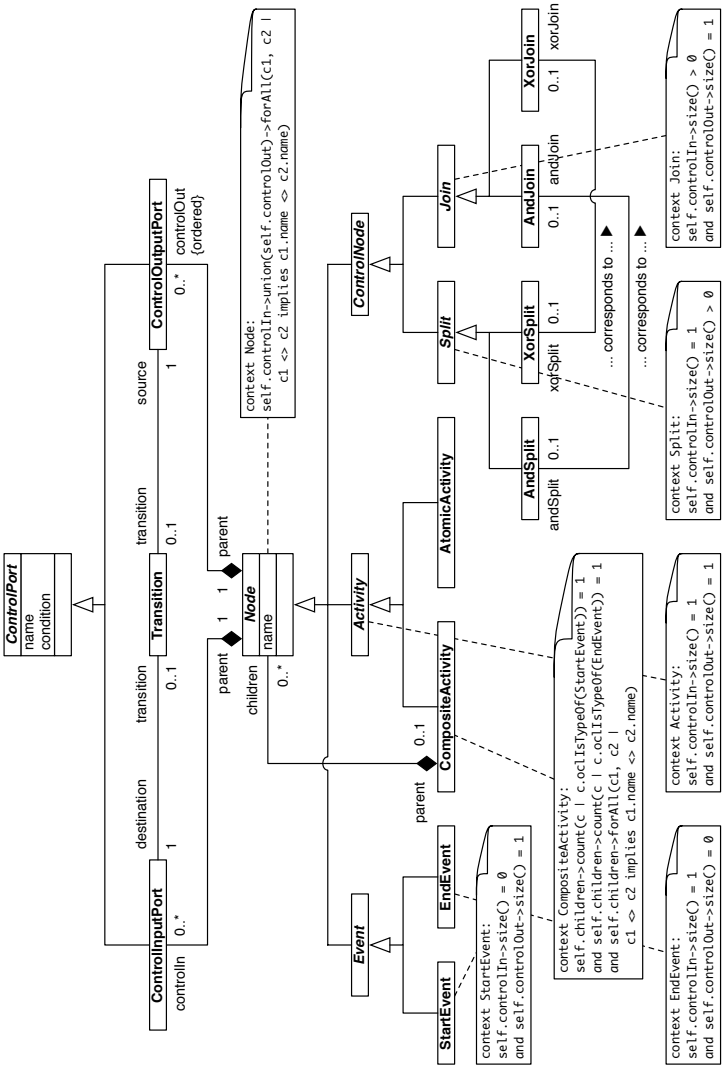


Figure 4.2: The UNIFY base language meta-model

Basic workflow pattern	Corresponding UNIFY construct(s)
WCP-1. Sequence	A Transition that connects a source Activity to a destination Activity
WCP-2. Parallel split	An AndSplit with its incoming Transition and outgoing Transitions
WCP-3. Synchronization	An AndJoin with its incoming Transitions and outgoing Transition
WCP-4. Exclusive choice	An XorSplit with its incoming Transition and outgoing Transitions
WCP-5. Simple merge	An XorJoin with its incoming Transitions and outgoing Transition

Table 4.2: Mapping from basic workflow patterns (Russell et al., 2006a) to corresponding UNIFY constructs

Using UNIFY's base language, the different concerns that would otherwise have to be specified in a single, monolithic workflow, can be implemented as separate modules.[2] In the example from Figure 4.1, the concerns are, among others, *order handling*, *book selection*, *payment*, *shipping*, *preference saving*, *reporting*, and *bank account verification*. Figure 4.3 illustrates how these concerns could be specified separately. Note that each of the composite activities in Figure 4.3 contains less activities than the corresponding composite activity in Figure 4.1.

In Section 4.3.2, we describe how the control flow perspective of concerns can be augmented with a data perspective. In Sections 4.4 and 4.5, we describe how independently specified concerns can subsequently be connected to each other.

4.3.2 Data Perspective

In Section 4.3.1, we introduced the control flow perspective of UNIFY's workflows. In this section, we introduce UNIFY's default data perspective, which is layered on top of the control flow perspective. Existing research (Russell et al., 2004a) has identified the following approaches for passing data from one activity to another:

- **Integrated control and data channels.** In this approach, control flow and data are passed simultaneously between activities, and transitions are annotated with which data elements must be passed. Activities can only access data that has been passed to them by an incoming transition.

- **Distinct control and data channels.** In this approach, data is passed between activities via explicit data links that are *distinct* from control flow links (i.e., transitions). Activities can only access data that has been passed to them by an incoming data link.

[2] Deciding which concerns should be modularized is partly a matter of personal preference, and is not the focus of our research.

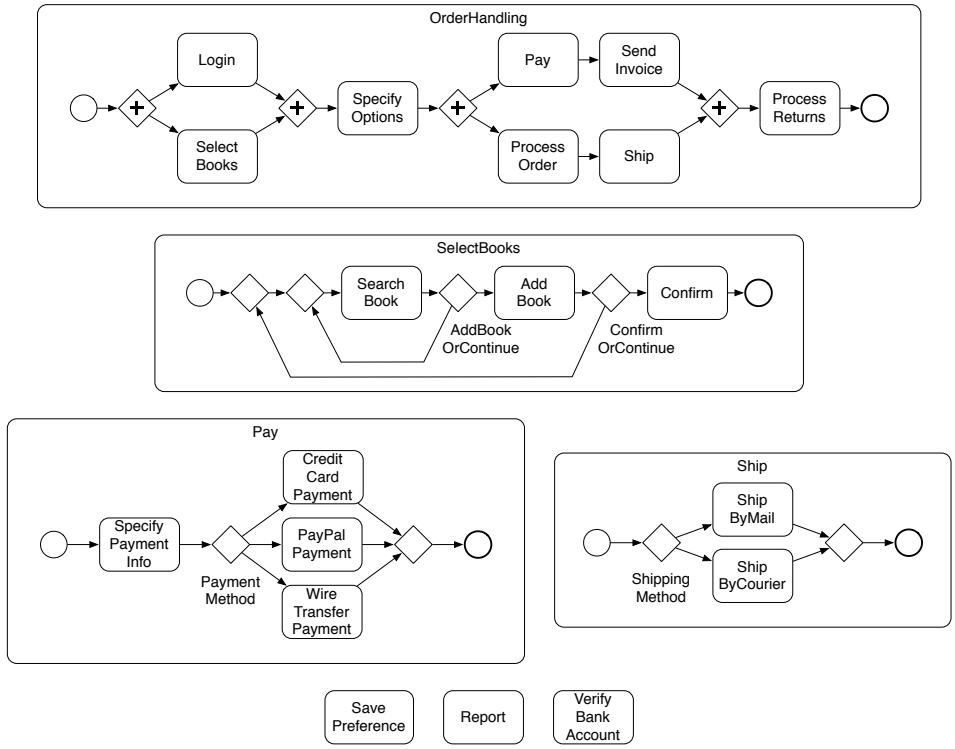


Figure 4.3: Independently specified workflow concerns

- **No data passing.** In this approach, activities share the same data elements, typically via access to some common scope. Thus, no explicit data passing is required. Activities can only access data that has been defined in a surrounding scope.

The *no data passing* approach is the most popular approach in current workflow languages, and most notably in BPEL. For example, BPEL activities can be grouped using the <scope> activity, which allows defining a number of variables that can be accessed from within the scope. Scopes can be nested in order to define elaborate hierarchies.

In the context of our instantiation towards WS-BPEL, we have defined an extension to the UNIFY meta-model for the *no data passing* approach. This extension is provided in Figure 4.4, and encompasses associating every CompositeActivity with a Scope, which defines any number of Variables. These Variables can then be manipulated using a number of AtomicActivities. Scopes form a hierarchy that mimics the composition hierarchy of their corresponding CompositeActivities. We do not foresee any fundamental obstacles to defining extensions to the UNIFY meta-model for the other two approaches.

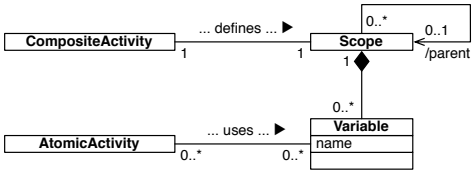


Figure 4.4: The UNIFY data meta-model for the *no data passing* approach

4.4 A Coherent Collection of Workflow-Specific Concern Connection Patterns

4.4.1 Existing Workflow Patterns

After identifying a number of workflow concerns and modeling them using the CompositeActivity construct, these different concerns need to be composed according to certain patterns. However, merely supporting patterns that correspond to the traditional aspect-oriented advices (*before*, *after*, and *around*) does not suffice, as these are all inherently sequential, whereas the workflow paradigm focuses heavily on parallelism and choice. Therefore, we need a more elaborate collection of concern connection patterns that recognizes the specific characteristics of workflows. The goal of this section is to identify a coherent set of such patterns.

Note that our focus on connection and composition of independently modularized workflow concerns is different than that of existing research on workflow patterns, such as the research of the Workflow Patterns initiative (cf. Section 2.1.2). For example, a workflow language that supports certain advanced control flow patterns may facilitate expressing certain workflows due to the increased expressivity of the language, but the workflows that are thus created are not necessarily more modular, as few of these patterns deal with modularization of concerns. The relation between the Workflow Patterns initiative's existing patterns and modularization of concerns is as follows:

1. None of the control flow patterns (Russell et al., 2006a) have a direct relation to modularization of concerns: the patterns document different generic, recurring constructs for specifying a workflow's control flow, but do not address how a workflow might be divided into different modules.

2. Some of the data patterns (Russell et al., 2004a) are related to modularization of concerns in the sense that they refer to the *structure* of a workflow: an executing instance of a workflow is called a *process instance* or *case*, which consists of *task instances*, which are either *atomic tasks*, *block tasks*, *multi-instance tasks*, or *multi-instance block tasks*. Block tasks and multi-instance block tasks correspond to the execution of a certain sub-workflow. Thus, the data patterns assume a certain hierarchical decomposition of workflows into sub-workflows. The individual data patterns deal with data visibility, data interaction, data transfer, and data-based routing with regard to each of the above structural elements of a workflow.

Our research goes beyond the purely hierarchical modularization that is assumed in this existing research by supporting the specification of unanticipated concern connections.

3. None of the resource patterns (Russell et al., 2004b) have a direct relation to modularization of concerns: they deal with creating work items that represent the work to be done during the execution of certain tasks, and subsequently organizing the execution of these work items, regardless of the modularization of the workflow to which the tasks belong.

4. The way in which exceptions are handled within a certain workflow is defined by its exception handling perspective. The exception handling patterns (Russell et al., 2006b) describe different ways of handling certain types of exceptions. Similar to the data perspective, a hierarchical decomposition of workflows into sub-workflows is assumed. Each task (including block tasks) can be associated with exception handling strategies for the exception types that may arise during the task's execution. Although exception handling can be considered a different concern than the workflow's base concern, workflow languages typically require specifying exception handling as part of the same module to which the exception handling concern applies.

5. The presentation patterns (La Rosa et al., 2011a,b) deal with presenting workflows in a way that promotes understandability by managing complexity. The *vertical modularization* pattern refers to hierarchical decomposition of workflows into sub-workflows. The *horizontal modularization* pattern refers to the partitioning of workflows into separate workflows that are executed in parallel, with synchronization being accomplished using message exchange between the parallel workflows. The *orthogonal modularization* pattern refers to modularization of crosscutting concerns using aspect-oriented mechanisms. UNIFY can be seen as an approach that aims to facilitate the realization of the *vertical modularization* and *orthogonal modularization* patterns. However, these presentation patterns are only concerned with *presenting* this kind of modularization, and not with realizing it within a workflow, e.g., using the most suitable advices.

Thus, none of the above patterns are well suited for specifying how independently modularized workflow concerns may be connected, and the remainder of this section is dedicated to identifying a coherent collection of workflow-specific concern connection patterns.

4.4.2 Outline of Our Proposal

Before we commence with the description of our actual concern connection patterns, we must briefly discuss the locations where different concerns can be connected to each other. In aspect-oriented terminology, these locations are called joinpoints. Similar to existing aspect-oriented workflow approaches such as AO4BPEL, we advocate the use of workflow activities as joinpoints, because workflow activities constitute the "units of

work" to be performed by the workflow, and other workflow elements such as control nodes and transitions merely serve to define the correct ordering of activities.

In some workflows, however, a certain unit of work may not have been modeled as a single (composite) activity. For example, an order handling workflow may not contain a single payment activity, but may instead contain a sequence of an activity that sends a payment request and an activity that processes a customer's payment. We advocate that such groups of workflow elements can serve as joinpoints, too. However, not any group of workflow elements makes sense in this regard: we require a certain form of well-formedness within this group. We find this well-formedness in the notion of *single-entry single-exit (SESE) fragments* proposed by Vanhatalo et al. (2007). Intuitively, a SESE fragment is a subgraph of a workflow graph which, for the purposes of workflow decomposition, could be replaced by a single activity. More formally, a SESE fragment is defined as follows (Vanhatalo et al., 2007):

Definition 1. *Let $G = (N, E)$ be a workflow graph. A* SESE fragment *(*fragment *for short) $F = (N', E')$ is a nonempty subgraph of G, i.e., $N' \subseteq N$ and $E' = E \cap (N' \times N')$ such that there exist edges $e_i, e_o \in E$ with $E \cap ((N \setminus N') \times N') = \{e_i\}$ and $E \cap (N' \times (N \setminus N')) = \{e_o\}$; e_i and e_o are called the entry and the exit edge of F, respectively.*

Note that a single workflow activity is always a fragment. Thus, by employing fragments as joinpoints, we support both single activities and well-formed groups of workflow elements as joinpoints.

In order to guide the description of our concern connection patterns, we distinguish between two main categories of patterns based on whether the connected concerns' behavior is inserted *outside* of joinpoint fragments or *inside* of joinpoint fragments. We will call the former category *external* concern connection patterns and the latter *internal* concern connection patterns. For each of our patterns, we provide a motivation and an example of its use. We also provide a textual description of the pattern, and illustrate its weaving using a figure. The way in which our patterns are realized in existing workflow languages such as AO4BPEL, the approach by Courbis and Finkelstein, and PADUS is discussed later in this chapter, in Section 4.4.5. The way in which our patterns are realized in UNIFY's connector mechanism is provided in Section 4.5, and the weaving of UNIFY's connectors is specified formally in Chapter 5.

4.4.3 External Concern Connection Patterns

External concern connection patterns are used to introduce behavior *outside* of joinpoint fragments. Within this category of patterns, we distinguish between four subcategories, which correspond to the four basic kinds of control flow structures that are prevalent in control flow patterns research and structured workflow languages such as BPEL. These sub-categories are:

1. **Sequential concern connection patterns**, which express that concerns are to be composed in sequence. We identify four patterns within this sub-category, i.e., the *before*, *after*, *replace*, and *around* concern connection patterns.

2. **Parallel concern connection patterns**, which express that concerns are to be composed in parallel with each other. We identify one pattern within this sub-category, i.e., the *parallel* concern connection pattern.

3. **Conditional concern connection patterns**, which express that concerns are to be composed as an alternative to each other. We identify one pattern within this sub-category, i.e., the *alternative* concern connection pattern.

4. **Iterating concern connection patterns**, which express that concerns are to be composed with each other in an iteration. We identify one pattern within this sub-category, i.e., the *iterating* concern connection pattern.

We will now discuss each of these seven patterns in detail.

4.4.3.1 The "Before" Concern Connection Pattern

Motivation Specifying that a given concern is to be executed *before* certain fragments in a base workflow.

Example In an order handling workflow, a logging activity is to be executed before every activity.

Description Activity A and composite activity B have been independently modularized.[3] During the execution of composite activity B, activity A is to be executed before each member fragment of a joinpoint set JP. Figure 4.5 illustrates the weaving of activity A before a fragment JP_x in composite activity B: activity A is inserted in composite activity B between fragment JP_x and its incoming transition t. Note that in our concern connection pattern diagrams, black rounded rectangles represent the advice activity, dashed clouds represent pieces of workflow that are not relevant to the description of the current pattern, gray rounded rectangles represent the joinpoint *fragment*, and dashed arrows represent transitions that are not relevant to the description of the current pattern. Although the workflows in our diagrams typically contain two clouds to represent pieces of workflow that are not relevant to the description of the pattern, these two pieces are not necessarily disjoint. In Figure 4.5, for example, composite activity B may contain paths from its start event to its end event that bypass fragment JP_x.

4.4.3.2 The "After" Concern Connection Pattern

Motivation Specifying that a given concern is to be executed *after* certain fragments in a base workflow.

Example In an order handling workflow, a reporting activity is to be executed after activities that handle payment or shipment of books.

[3] Activity A can be either atomic or composite. Activity B must be composite.

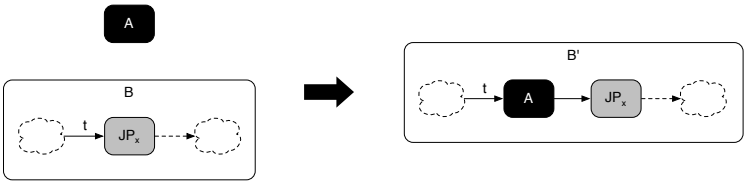


Figure 4.5: The “before” concern connection pattern

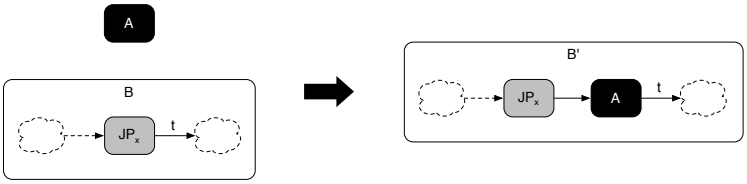


Figure 4.6: The “after” concern connection pattern

Description Activity A and composite activity B have been independently modularized. During the execution of composite activity B, activity A is to be executed after each member fragment of a joinpoint set JP. Figure 4.6 illustrates the weaving of activity A after a fragment JP_x in composite activity B: activity A is inserted in composite activity B between fragment JP_x and its outgoing transition t.

4.4.3.3 The “Replace” Concern Connection Pattern

Motivation Specifying that a given concern is to be executed *instead of* certain fragments in a base workflow. This pattern can be used to adapt an existing base workflow for use in different contexts, but it can also be used to achieve hierarchical decomposition by connecting abstract activities within a base workflow to concrete activities that are modularized separately.

Example In an order handling workflow, a default payment activity may need to be replaced by a variant before deploying the workflow for use by customers in another country.

Description Activity A and composite activity B have been independently modularized. During the execution of composite activity B, activity A is to be executed instead of each member fragment of a joinpoint set JP. Figure 4.7 illustrates the weaving of activity A instead of a fragment JP_x in composite activity B: activity A is inserted in composite activity B instead of fragment JP_x.

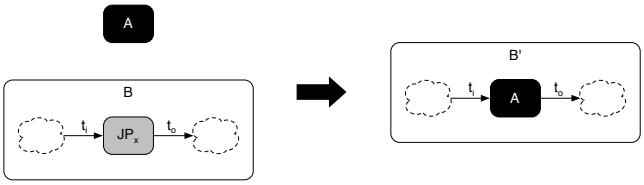


Figure 4.7: The "replace" concern connection pattern

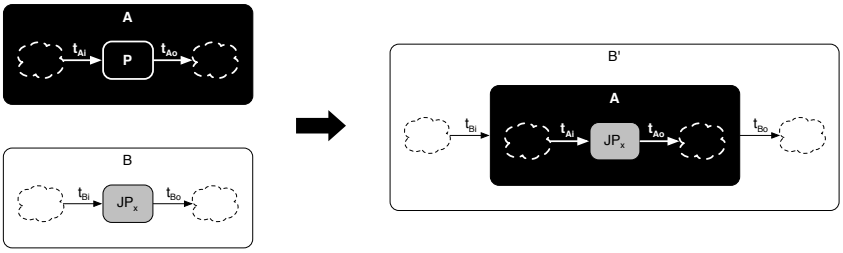


Figure 4.8: The "around" concern connection pattern

4.4.3.4 The "Around" Concern Connection Pattern

Motivation Specifying that a given concern is to be executed *around* certain fragments in a base workflow. The concern may or may not cause the fragment to which it is applied to be skipped.

Example In an order handling workflow, some authentication concern may need to be applied to the activity that allows selecting books. The authentication concern will control whether this activity is actually executed.

Description Composite activity A and composite activity B have been independently modularized. Composite activity A contains a proceed activity P. During the execution of composite activity B, composite activity A is to be executed instead of each member fragment of a joinpoint set JP. When, during the execution of composite activity A, proceed activity P is encountered, the member fragment is to be executed. Figure 4.8 illustrates the weaving of activity A around a fragment JP_x in composite activity B: activity A is inserted in composite activity B instead of fragment JP_x, and within activity A, proceed activity P is replaced by fragment JP_x. Remember from Section 4.4.3.1 that the pieces of workflow represented in our diagrams by clouds are not necessarily disjoint. In Figure 4.8, composite activity A may thus contain paths from its start event to its end event that bypass proceed activity P.

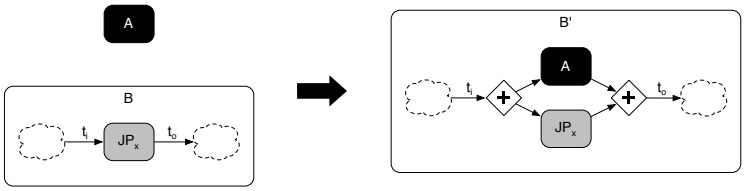


Figure 4.9: The "parallel" concern connection pattern

4.4.3.5 The "Parallel" Concern Connection Pattern

Motivation Specifying that a given concern is to be executed *in parallel with* certain fragments in a base workflow.

Example In an order handling workflow, an activity that records user preferences may need to be executed in parallel with activities that retrieve book information or add books to the user's shopping basket.

Description Activity A and composite activity B have been independently modularized. During the execution of composite activity B, activity A is to be executed in parallel with each member fragment of a joinpoint set JP. Figure 4.9 illustrates the weaving of activity A in parallel with a fragment JP_x in composite activity B: an AND-split is inserted between fragment JP_x and its incoming transition t_i, and an AND-join is inserted between fragment JP_x and its outgoing transition t_o, causing fragment JP_x to act as a parallel branch of the thus created control structure. Activity A is inserted as the second parallel branch of this control structure.

4.4.3.6 The "Alternative" Concern Connection Pattern

Motivation Specifying that a given concern is to be executed *as an alternative to* certain fragments in a base workflow.

Example In an order handling workflow, a different payment activity than the default one may need to be executed if the user is underaged.

Description Activity A and composite activity B have been independently modularized. During the execution of composite activity B, activity A is to be executed as an alternative to each member fragment of a joinpoint set JP if condition c is satisfied. Figure 4.10 illustrates the weaving of activity A as an alternative to a fragment JP_x in composite activity B: an XOR-split is inserted between fragment JP_x and its incoming transition t_i, and an XOR-join is inserted between fragment JP_x and its outgoing transition t_o, causing fragment JP_x to act as an alternative branch of the thus created control structure. Activity A is inserted as the second alternative branch of this control structure. Condition c determines which of the alternative branches is taken.

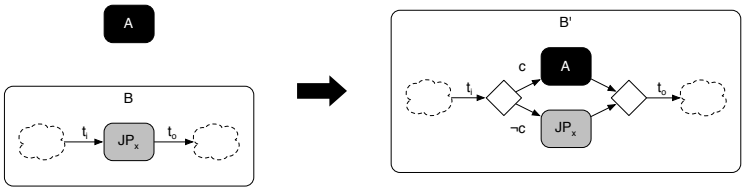


Figure 4.10: The "alternative" concern connection pattern

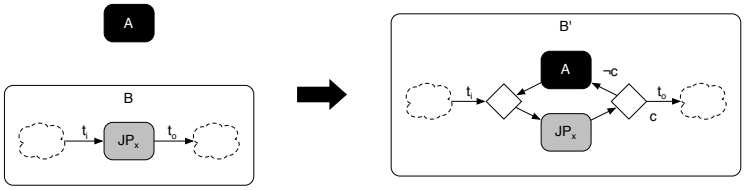


Figure 4.11: The "iterating" concern connection pattern

4.4.3.7 The "Iterating" Concern Connection Pattern

Motivation Specifying that a given concern is to be executed *by iterating over* certain fragments in a base workflow.

Example In an order handling workflow, an activity that handles a failed login attempt may be executed in an iteration with the existing login activity until login succeeds, in order to implement the business's account security policy.

Description Activity A and composite activity B have been independently modularized. During the execution of composite activity B, activity A is to be executed in iterations with each member fragment of a joinpoint set JP until condition c is satisfied. Figure 4.11 illustrates the weaving of activity A in an iteration with a fragment JP_x in composite activity B: an XOR-split is inserted between fragment JP_x and its outgoing transition t_o, and an XOR-join is inserted between fragment JP_x and its incoming transition t_i, causing fragment JP_x to act as the forward branch of the thus created cycle. Activity A is inserted as the returning branch of this cycle. The cycle will continue as long as condition c is false.

Note that, by inserting the advice activity A as the returning branch of the cycle, we obtain a control structure that is slightly different from traditional while or repeat control structures, in which the advice and the joinpoint fragment JP_x might be expressed together in the control structure's body. Our approach has the advantage that the advice is not executed when condition c is satisfied after the initial execution of the joinpoint, and the advice is executed exactly once between all subsequent executions of the joinpoint. This makes the advice suited for preparing the workflow for a next iteration, or giving the user feedback on a previous iteration.

This concludes our description of the seven external concern connection patterns. We will now describe the other main category of concern connection patterns, i.e., internal concern connection patterns.

4.4.4 Internal Concern Connection Patterns

Internal concern connection patterns are used to introduce behavior *inside* of joinpoint fragments. More specifically, they introduce additional control flow dependencies between different nodes within a fragment. In order to control the effects of these patterns, we restrict the allowed joinpoint fragments to (parallel or conditional) *control structures*, i.e., the patterns introduce additional control flow dependencies within fragments that either start with an AND-split and end with that split's corresponding AND-join, or start with an XOR-split and end with that split's corresponding XOR-join. We propose two patterns: the first allows synchronizing branches of a parallel control structure, while the second allows switching branches of a conditional control structure.

4.4.4.1 The "Synchronizing Parallel Branches" Concern Connection Pattern

Motivation Specifying that a given concern is to perform a synchronization between two branches in an existing parallel control structure of a base workflow.

Example In an order handling workflow, payment and shipping may be performed in parallel branches. Depending on the type of customers that is expected to use the workflow, a verification activity may need to synchronize both branches, i.e., prevent shipment before payment of the order has been verified.

Description Activity A and composite activity B have been independently modularized. Composite activity B contains a fragment JP that is a parallel control structure.[4] Activity A is to be executed after a fragment JP_s (s stands for *split*) has been executed and before a fragment JP_j (j stands for *join*) is executed in order to synchronize two branches of the control structure. Figure 4.12 illustrates the weaving of activity A in composite activity B: a new AND-split is inserted between fragment JP_s and its outgoing transition t_o, a new AND-join is inserted between fragment JP_j and its incoming transition t_i, and activity A is placed between the inserted AND-split and AND-join. Note that this implies that the resulting workflow will no longer be structured.

4.4.4.2 The "Switching Alternative Branches" Concern Connection Pattern

Motivation Specifying that a given concern is to perform a switch between two branches in an existing conditional control structure of a base workflow.

[4] In these internal concern connection patterns, we no longer assume the existence of a set of joinpoints, but rather restrict our description to a single joinpoint, as it is unlikely that the exact same synchronization or switching will need to be performed at multiple locations within a workflow.

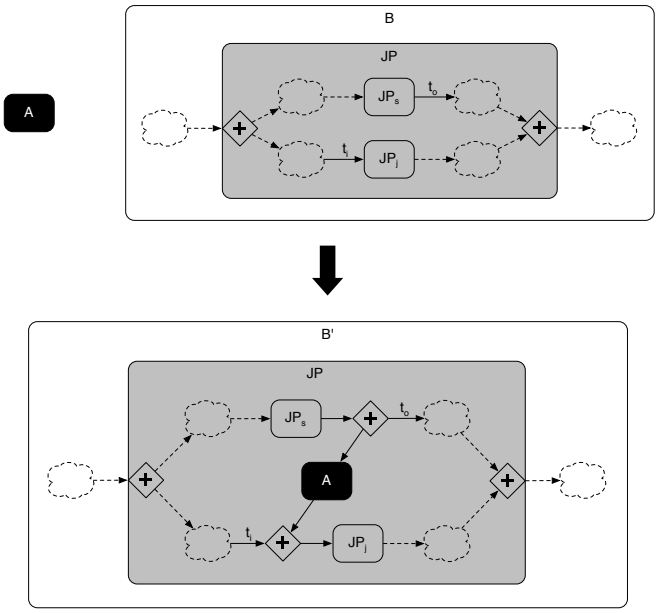


Figure 4.12: The "synchronizing parallel branches" concern connection pattern

Example In an order handling workflow, shipment of a customer's order by courier and retrieval of an order by a customer at the store may be performed in alternative branches. A dedicated activity could allow switching between both branches in order to support situations where customers unexpectedly show up at the store to retrieve orders that have not yet been handed over to a courier.

Description Activity A and composite activity B have been independently modularized. Composite activity B contains a fragment JP that is a conditional control structure. Activity A is to be executed after a fragment JP_s has been executed if condition c is satisfied, and before a fragment JP_j is executed, in order to switch between two branches of the control structure. Figure 4.13 illustrates the weaving of activity A in composite activity B: a new XOR-split is inserted between fragment JP_s and its outgoing transition t_o, a new XOR-join is inserted between fragment JP_j and its incoming transition t_i, and activity A is placed between the inserted XOR-split and XOR-join, with condition c determining whether control flow will be switched towards activity A or not. Note that this implies that the resulting workflow will no longer be structured.

4.4.5 Realization of Concern Connection Patterns in Existing Approaches

The "before" and "after" concern connection patterns are natively implemented by the *before* and *after* advices offered by traditional aspect-oriented workflow languages such as AO4BPEL, the approach by Courbis and Finkelstein, and PADUS. Similarly, the "around"

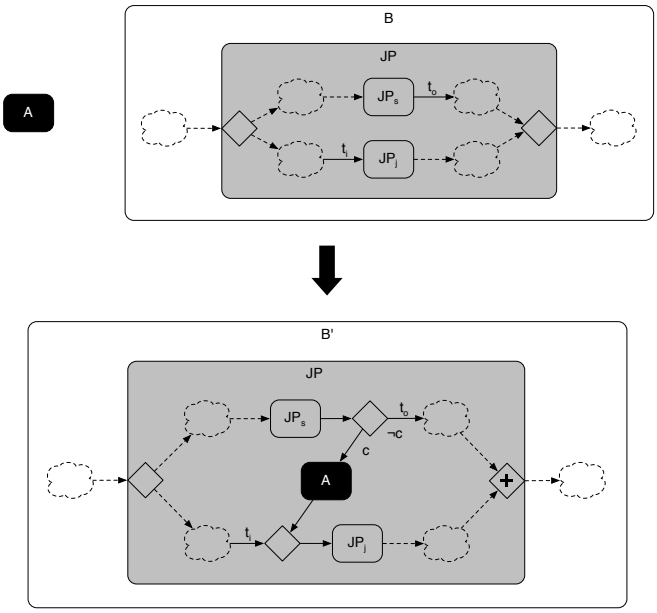


Figure 4.13: The “switching alternative branches” concern connection pattern

concern connection pattern is natively implemented by the *around* advice offered by AO4BPEL and PADUS (but which is not supported by Courbis and Finkelstein's approach). In fact, all external concern connection patterns can be implemented using the *around* advice:

- The “before” pattern can be implemented using an *around* advice whose body is a sequence of the actual advice behavior and a proceed activity.
- The “after” pattern can be implemented using an *around* advice whose body is a sequence of a proceed activity and the actual advice behavior.
- The “replace” pattern can be implemented using an *around* advice whose body does not contain a proceed activity.
- The “parallel” pattern can be implemented using an *around* advice whose body is a parallel control structure of which one branch is the actual advice behavior and another branch is a proceed activity.
- The “alternative” pattern can be implemented using an *around* advice whose body is an alternative control structure of which one branch is the actual advice behavior and another branch is a proceed activity.
- The “iterating” pattern can be implemented using an *around* advice whose body is a sequence of a proceed activity and a while control structure that iterates over a sequence of the actual advice behavior and a proceed activity.

Category	Pattern
External — sequential patterns	CCP-1. Before
	CCP-2. After
	CCP-3. Replace
	CCP-4. Around
External — parallel patterns	CCP-5. Parallel
External — conditional patterns	CCP-6. Alternative
External — iterating patterns	CCP-7. Iterating
Internal patterns	CCP-8. Synchronizing parallel branches
	CCP-9. Switching alternative branches

Table 4.3: Concern connection patterns

However, using the *around* advice to express all these different patterns negatively affects possible reuse of the advice behavior. For example, (1) when using an *around* advice to express the "before" pattern, the advice body is a sequence of the actual advice behavior and a proceed activity. Thus, this advice cannot be reused in situations where another pattern is required because the sequence and the proceed activity are hard-coded into the advice body, whereas they are conceptually part of the connection logic. (2) When using an *around* advice to express the "parallel" pattern, the advice cannot be reused in situations where another pattern is required because the parallel control structure and the proceed activity are hard-coded into the advice body, whereas they are conceptually part of the connection logic. (3) When using an *around* advice to express the "alternative" pattern, the advice cannot be reused in situations where another pattern is required because the conditional control structure — including the condition that determines which branch is taken — and the proceed activity are hard-coded into the advice body, whereas they are conceptually part of the connection logic. Therefore, UNIFY makes a very clear distinction between the actual advice behavior, which is implemented as a CompositeActivity, and the connection logic, which will be implemented as a Connector. We believe that the concern connection patterns identified above form a good basis for developing such a connector mechanism.

4.4.6 Conclusions

In this section, we have introduced a total of nine patterns according to which independently modularized workflow concerns can be connected. These patterns are summarized in Table 4.3. Although we do not claim that this is an exhaustive list of all concern connection patterns that could ever be useful in a workflow context, we do think that our patterns form a coherent collection that can be used to connect most workflow concerns in an effective way. The patterns go beyond the traditional *before, after* and *around* advices that are common in traditional aspect-oriented programming by recognizing the prevalence of parallel, conditional and iterating control structures in workflows.

4.5 Connector Mechanism

In this section, we describe the UNIFY connector mechanism, which defines how workflow concerns that have been implemented using the UNIFY base language can be connected to each other. The UNIFY connector mechanism is based on aspect-oriented principles (Kiczales et al., 1997). Therefore, we will describe it using the same template we used in Chapters 2 and 3, i.e., the template proposed in AOSD-Europe's survey on aspect-oriented programming languages by Brichau and Haupt (2005). Our connector mechanism is defined using a meta-model that complements the base language meta-model of Section 4.3. This connector language meta-model is shown in Figure 4.14, and we will refer to this meta-model throughout our description.

4.5.1 Joinpoint Model and Pointcut Language

A UNIFY workflow is created by specifying a number of workflow concerns as independent CompositeActivities, and subsequently composing these in a workflow composition. To this end, we introduce connectors that specify how two CompositeActivities are to be composed. The behavior implemented by one CompositeActivity can be introduced into another CompositeActivity at certain nodes, with the latter CompositeActivity being oblivious of this possible introduction of behavior. Thus, the former CompositeActivity can be considered an *advice*, the latter CompositeActivity can be considered a *base workflow*, and the locations where the former's behavior is introduced into the latter can be considered *joinpoints*.[5]

Remember that **joinpoints** are well-defined points within the specification of a concern where extra functionality — the advice — can be inserted. Joinpoints in existing aspect-oriented approaches for workflows are either the XML elements of the process definition (as in AO4BPEL or the approach by Courbis and Finkelstein) or the workflow's activities (as in PADUS). UNIFY goes beyond these existing approaches by not only supporting workflow activities as joinpoints, but also certain groups of workflow elements, more specifically *SESE fragments* (*fragments* for short), as defined in Section 4.4.2. This allows, for example, introducing an advice around a group of activities in a base workflow. Because single activities are merely a specific kind of fragment, fragments are the only kind of joinpoints supported by UNIFY.

The joinpoint model is static, which has the advantage of allowing us to define a clear weaving semantics using the *Petri nets* and *Graph Transformation* formalisms. UNIFY's semantics is discussed in detail in Chapter 5.

Pointcuts are expressions that resolve to a set of joinpoints, and are used to specify where in the base workflow a connector should add its functionality. Because all workflow Nodes have names that are unique among their siblings, every Node can be uniquely identified by prepending the name of the Node with the names of its ancestors (we will call this the *canonical name* of the Node). Because a workflow Fragment has a single entry Node and a single exit Node, a Fragment can be uniquely identified

[5] In general, the advice can be any kind of Activity. In realistic situations, however, it will be a CompositeActivity as described in this paragraph.

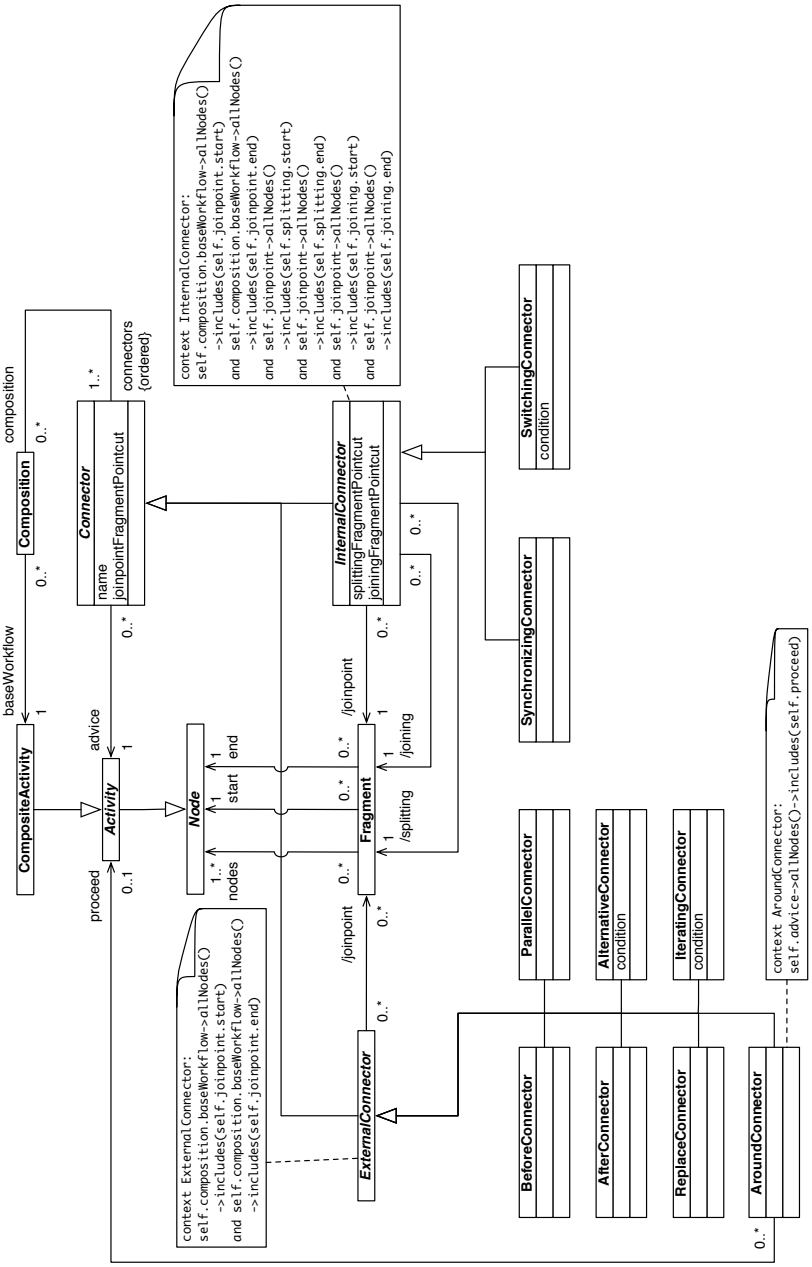


Figure 4.14: The UNIFY connector language meta-model

Fragment pointcuts	Description
`fragment(EntryIdentifier, ExitIdentifier)`	Selects the Fragment which starts at the Node specified by the first identifier, and which ends at the Node specified by the second identifier. If no such Fragment exists, an error is raised.

Activity pointcuts	Description
`activity(IdentifierPattern)`	Selects any Activity whose name matches the identifier pattern.
`compositeActivity(IdentifierPattern)`	Selects any CompositeActivity whose name matches the identifier pattern.
`atomicActivity(IdentifierPattern)`	Selects any AtomicActivity whose name matches the identifier pattern.

Table 4.4: Pointcut predicates

by the canonical names of its entry and exit Nodes. As is shown in Table 4.4, we provide the `fragment(EntryIdentifier, ExitIdentifier)` pointcut predicate for selecting Fragments in this way.

Single Activities can be selected using the `fragment` predicate with the canonical name of the activity as both the first and second argument of the predicate. However, because Activities are the most important kind of workflow element, we provide a number of shorthand pointcut predicates for selecting single Activities, which eliminate the need for this kind of repetition. Because the arguments of these pointcut predicates are actually identifier patterns that are matched to the canonical names of the Activities within a workflow, they allow selecting *sets of* single Activities. Thus, UNIFY supports *quantification* in Activity pointcuts.

Note that our Fragment pointcuts do *not* support quantification because of the complexity of selecting sets of Fragments using a textual syntax: merely extending the `fragment(EntryIdentifier, ExitIdentifier)` pointcut predicate towards identifier patterns would not suffice, as both identifier patterns would resolve to a set of nodes, while it would be difficult to determine which entry node corresponds to which exit node. In order to support quantification in Fragment pointcuts, we essentially need a means of specifying *fragment patterns* rather than identifier patterns. Existing research (Förster et al., 2007) allows specifying patterns of workflow nodes in UML activity diagrams using a visual language. Although this language is used to specify process constraints, it could be used for the purpose of specifying pointcuts as well. Other existing research, by one of our master students (Gheysels, 2007; Braem and Gheysels, 2007), extended the pointcut language of PADUS with support for specifying protocols, and extended the advice model and language of PADUS with support for *stateful aspects* by allowing to attach an advice to a protocol. Patterns of workflow nodes, such as those offered by Förster et al. (2007), could be used to specify protocols as well. We consider the extension of UNIFY with support for quantification in Fragment pointcuts and support for stateful aspects to be an interesting avenue of future work.

Unlike the PADUS pointcut language, the UNIFY pointcut language does not aim to provide a full logic language. PADUS introduced such a logic language in order to abstract

over the specific language constructs of BPEL processes. In UNIFY, the base language meta-model already abstracts over language constructs of specific workflow languages, and a more traditional pointcut language is thus sufficient for the purposes of UNIFY.

4.5.2 Advice Model and Language

In order to offer a more expressive advice model than existing aspect-oriented workflow approaches and go beyond the workflow-specific *in* advice offered by PADUS, UNIFY's advice model is based on the concern connection patterns we identified in Section 4.4. More specifically, we offer one advice type for each of the concern connection patterns. Thus, we obtain seven advice types for the external concern connection patterns, i.e., the **before**, **after**, **replace**, **around**, **parallel**, **alternative**, and **iterating** advice types, and two advice types for the internal concern connection patterns, i.e., the **synchronizing** and **switching** advice types. Additionally, *variants* of these advice types allow expressing combinations of sequential and parallel concern connection patterns with conditional and iterating patterns.

An essential characteristic of UNIFY is that advice code is not specified in a different kind of module than base code: just like the base workflow, the advice is modularized as a CompositeActivity. This makes UNIFY a *uniform* approach, as both aspect and base modules have the same form. This implies that UNIFY does not offer additional language constructs for use in advice modules, as is the case in traditional AOP approaches, e.g., to allow accessing the (runtime) context in which the advice is executed, or defining the location where the original joinpoint behavior should be executed in an *around* advice. Instead, this is accomplished using UNIFY's connector construct, which is discussed in Section 4.5.3. This approach is inspired by *symmetric AOP* approaches (Tarr et al., 1999; Suvée et al., 2006), which prevent the introduction of specialized aspect modules by removing the distinction between base concerns and advice concerns. However, we do not consider UNIFY a perfectly symmetric approach, because there is still a notion of base concerns and advice concerns within a given composition. We thus find the term *uniform* more appropriate.

4.5.3 Aspect Module Model

Early aspect-oriented approaches offered a single language construct for modularizing crosscutting concerns: the aspect. An aspect had two responsibilities: on the one hand, the aspect's advice body specified the modularized crosscutting behavior, while on the other hand, the aspect's advice type and pointcut specified how and where the behavior should be composed with the base program. This approach has the disadvantage that both responsibilities are tightly coupled, which precludes effective reuse of the modularized behavior in different composition contexts.

Several approaches have been developed that address the above problem. A first approach is employed by ASPECTJ (Kiczales et al., 2001), which allows specifying *abstract* aspects and pointcuts. By specifying crosscutting behavior in an abstract aspect which specifies an abstract pointcut, and augmenting the abstract aspect with different concrete pointcuts using *inheritance*, the abstract aspect's behavior can be reused in differ-

	Base code specified in ...	**Crosscutting code specified in ...**	**Composition logic specified in ...**
ASPECTJ	JAVA classes	Aspects (advice body)	Aspects (advice type and pointcut)
AO4BPEL	BPEL processes	Aspects (advice body)	Aspects (advice type and pointcut)
Courbis and Finkelstein	BPEL processes	Aspects (advice body)	Aspects (advice type and pointcut)
PADUS	BPEL processes	Aspects (advice body)	Aspects (advice type and pointcut)
JASCO	JAVA classes	Aspect beans	Connectors
UNIFY	CompositeActivities	CompositeActivities	Connectors

Table 4.5: Comparison of ASPECTJ, AO4BPEL, Courbis and Finkelstein, PADUS, JASCO, and UNIFY modularization approaches

ent sub-aspects. A second approach is employed by JASCO (Suvée and Vanderperren, 2003) and *aspectual components* (Lieberherr et al., 1999), and is inspired by component-based software development (CBSD; cf. Shaw and Garlan, 1996). In this approach, the crosscutting behavior is specified in a separate *aspect bean* (JASCO) or *aspectual component* (aspectual components), while its deployment is specified in a separate *connector*. Thus, an aspect bean or aspectual component's behavior can be reused in different connectors.

In UNIFY, we have introduced a uniform modularization mechanism, which allows expressing both regular and crosscutting behavior using the same language construct (i.e., the CompositeActivity). The way in which different CompositeActivities should be composed is specified in separate connectors. Connectors can specify both crosscutting concern connection patterns (such as the before, after and around advice types of JASCO), but also non-crosscutting connection patterns similar to the connection of components in CBSD. Because the composition logic is no longer specified in the same module as the crosscutting behavior, the crosscutting behavior can be reused effectively in different composition contexts. The differences between the modularization approaches of ASPECTJ, AO4BPEL, Courbis and Finkelstein's approach, PADUS, JASCO, and UNIFY are summarized in Table 4.5.

As we already mentioned above, UNIFY's connectors are used to allow accessing the (runtime) context in which the crosscutting behavior is executed, or defining the location where the original joinpoint behavior should be executed in an *around* advice, and thus support the uniform modularization mechanism. The former is achieved by augmenting the UNIFY data meta-model of Figure 4.4 with a DataMapping concept, as is shown in Figure 4.15. When a certain concern is connected to a base workflow using one of UNIFY's connectors, these data mappings will specify how the input and output data of the connected concern is mapped to the data of the base workflow. We have extended our connector syntax with such data mappings for BPEL workflows. In addition to these data mappings, the data perspective also arises within UNIFY's connectors when *conditions* are specified. However, in order to focus our discussion on the control

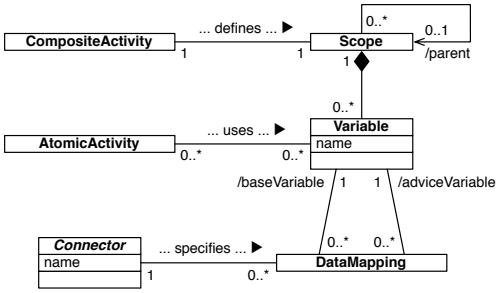


Figure 4.15: The augmented UNIFY data meta-model for the *no data passing* approach

flow perspective, we will abstract from the concrete data perspective of the connected workflows in the following description of UNIFY's connectors.

UNIFY offers two main kinds of connectors: ExternalConnectors and InternalConnectors (cf. the class hierarchy of the connector language meta-model of Figure 4.14). The former are described in Section 4.5.3.1, whereas the latter are described in Section 4.5.3.2.

4.5.3.1 External Connectors

ExternalConnectors implement the seven external concern connection patterns of Section 4.4.3. Using a one-to-one mapping from concern connection patterns to connectors, we obtain the before, after, replace, around, parallel, alternative, and iterating connectors.

Before Connectors BeforeConnectors specify that the advice Activity should be introduced into the base workflow *before* each of the joinpoints specified by the Connector's pointcut, which is represented in the connector language meta-model of Figure 4.14 as the joinpointFragmentPointcut attribute. Thus, these connectors implement the "before" concern connection pattern (CCP-1; cf. Section 4.4.3.1). For example, the before connector in Figure 4.16, which is named `LoggingConnector`, specifies that the `Logging` activity should be executed before each activity in the `OrderHandling` base workflow. As is illustrated by this example, we have opted for a declarative text-based syntax for our connector language, which is specified in Backus–Naur form in Appendix A. The first line of each connector indicates the name of the connector, while the connector itself starts with the `CONNECT` keyword followed by the name of the advice Activity. Next, one or more keywords indicate the concern connection pattern; in this example, the `BEFORE` keyword indicates the "before" concern connection pattern. A pointcut selects the joinpoint(s) to which the connector is to be applied. In our current implementation, the identifier pattern of the `activity(IdentifierPattern)` pointcut predicate

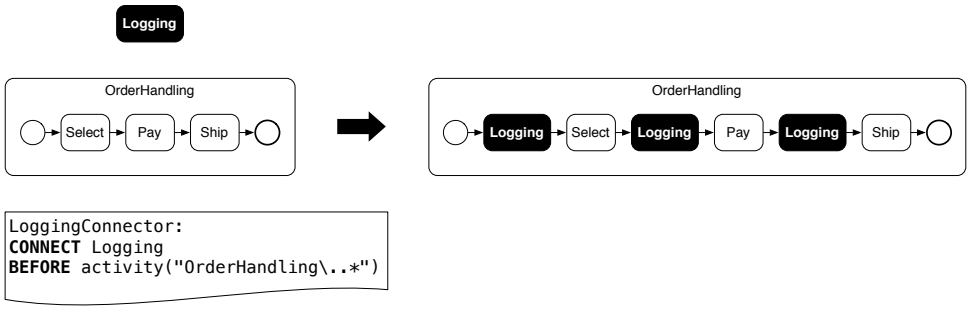


Figure 4.16: Example before connector

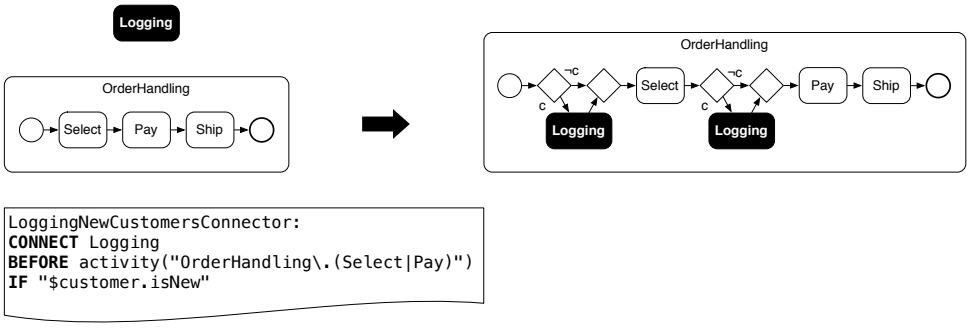


Figure 4.17: Example conditional before connector

is specified using JAVA's regular expression language.[6] In our examples, we use Activity pointcuts in order to illustrate the use of quantification, but Fragment pointcuts can be used just as well.

In order to facilitate expressing combinations of the "before" pattern with other patterns, we offer a conditional variant of this pattern. For example, in Figure 4.17, the connector named `LoggingNewCustomersConnector` specifies that the `Logging` activity should be executed before the `Select` and `Pay` activities within the `OrderHandling` base workflow *if the customer is a new customer*.[7] To this end, the connector syntax offers an optional *if* clause, which consists of the `IF` keyword followed by a condition that queries the data perspective of the workflow.[8] UNIFY does not impose a concrete language for expressing the conditions; the language used in our examples is XPATH.[9]

[6] Thus, the `OrderHandling\..*` pattern used in Figure 4.16 will match any identifier that starts with `OrderHandling`, followed by a single '.' character, followed by any number of arbitrary characters.

[7] The `OrderHandling\.(Select|Pay)` pattern used in Figure 4.17 will match any identifier that starts with `OrderHandling`, followed by a single '.' character, followed by either `Select` or `Pay`.

[8] In our figures, this condition is represented in BPMN diagrams as *c*.

[9] Thus, the `$customer.isNew` query refers to the `isNew` part of the `customer` variable.

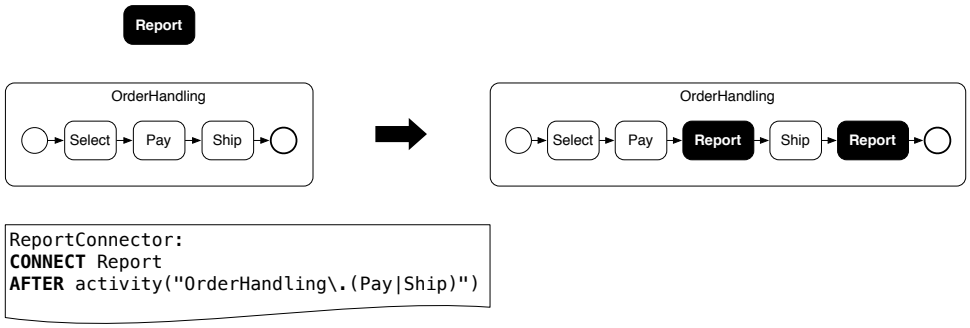


Figure 4.18: Example after connector

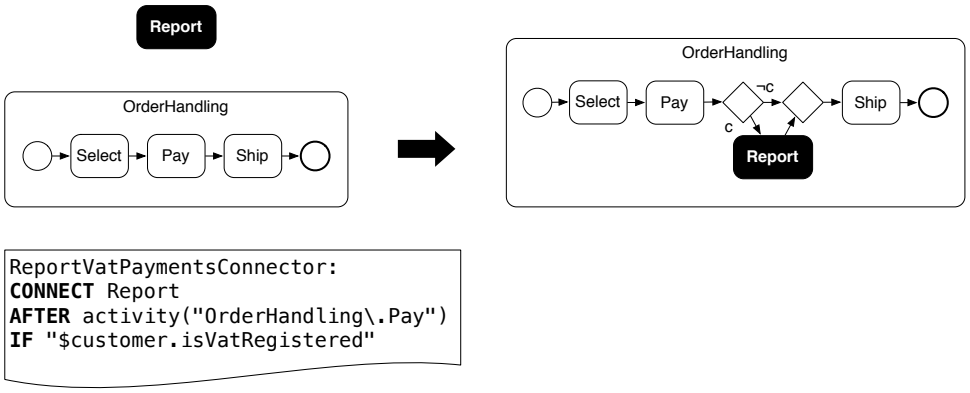


Figure 4.19: Example conditional after connector

After Connectors AfterConnectors specify that the advice Activity should be introduced into the base workflow *after* each of the joinpoints specified by the joinpointFragment-Pointcut. Thus, these connectors implement the "after" concern connection pattern (CCP-2; cf. Section 4.4.3.2). For example, the after connector in Figure 4.18, which is named `ReportConnector`, specifies that the `Report` activity should be executed after the `Pay` and `Ship` activities within the `OrderHandling` base workflow.

In order to facilitate expressing combinations of the "after" pattern with other patterns, we offer a conditional variant of this pattern. For example, in Figure 4.19, the connector named `ReportVatPaymentsConnector` specifies that the `Report` activity should be executed after the `Pay` activity within the `OrderHandling` base workflow *if the customer is registered for value added tax.*

Replace Connectors UNIFY's connectors, as well as the concern connection patterns on which they are based, invert the traditional passing of control from a main workflow into sub-workflows: they specify that a certain concern should be adapted, while this

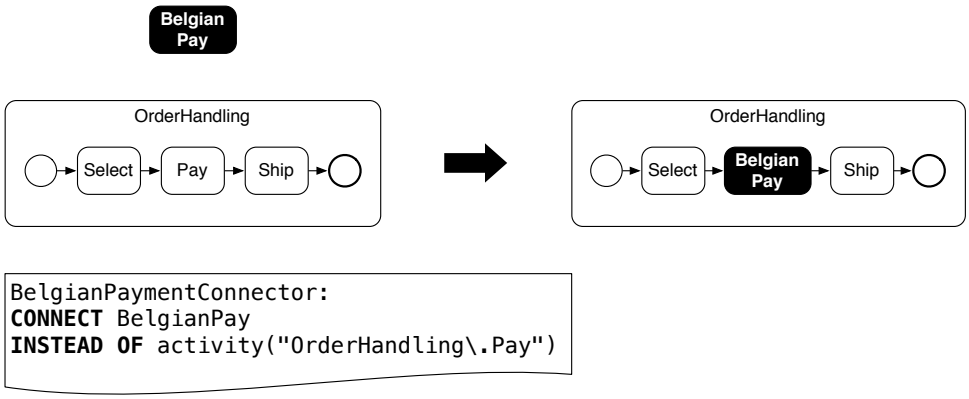
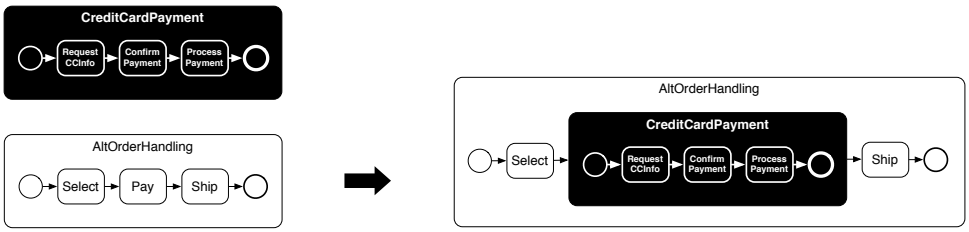


Figure 4.20: Example replace connector (inversion of control)

concern is not necessarily aware of this adaptation. In this way, connectors can be used to add concerns that were not anticipated when the concern to which they are applied was created, and thus constitute the constructs by means of which aspect-orientation is introduced into UNIFY. As we mentioned in Section 4.4.3.3, however, the "replace" pattern does not necessarily imply such inversion of control, as it can also be used to achieve hierarchical decomposition by connecting abstract activities within a base workflow to concrete activities that are modularized separately. Therefore, we will discuss two examples of ReplaceConnectors, which illustrate the two possible intentions with which ReplaceConnectors can be used.

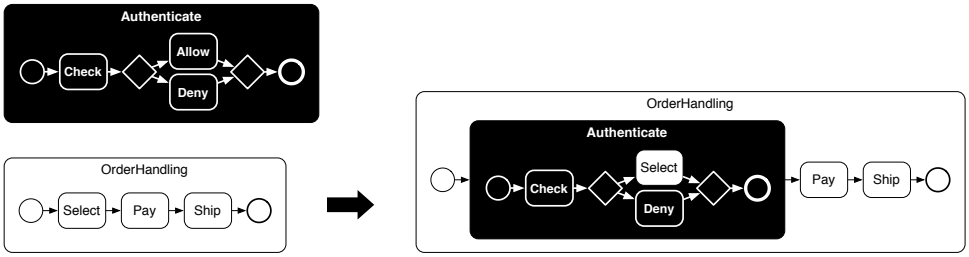
For example, consider the ReplaceConnector in Figure 4.20. It is applied to the `OrderHandling` base workflow, in which the `Pay` activity is assumed to be an activity that handles payment within the base workflow. The ReplaceConnector, which is called `BelgianPaymentConnector`, can be used to replace the `Pay` activity by the `BelgianPay` activity, for example because the base workflow is not sufficiently tailored towards Belgian customers. In this situation, the base workflow is not necessarily aware of possible adaptation by a connector.

As a second example, consider the ReplaceConnector in Figure 4.21. It is applied to the `AltOrderHandling` workflow, in which the `Pay` activity is assumed to be a placeholder activity (i.e., an Activity that is present in the base workflow, but performs no real function other than indicating that it should be replaced by another Activity at some point before execution). The ReplaceConnector, which is called `CCPaymentConnector`, specifies that the `Pay` activity in the `AltOrderHandling` composite activity should be implemented by executing the `CreditCardPayment` activity. In this situation, the `AltOrderHandling` workflow is aware of the future adaptation by a connector (because the workflow contains a placeholder activity). By specifying the link between the placeholder activity and the activity that implements its functionality in a separate connector instead of inside the CompositeActivity itself, we reduce coupling between the main concern and the sub-concern, thus promoting reuse.




```
CCPaymentConnector:
CONNECT CreditCardPayment
INSTEAD OF activity("AltOrderHandling\.Pay")
```

Figure 4.21: Example replace connector (hierarchical decomposition)

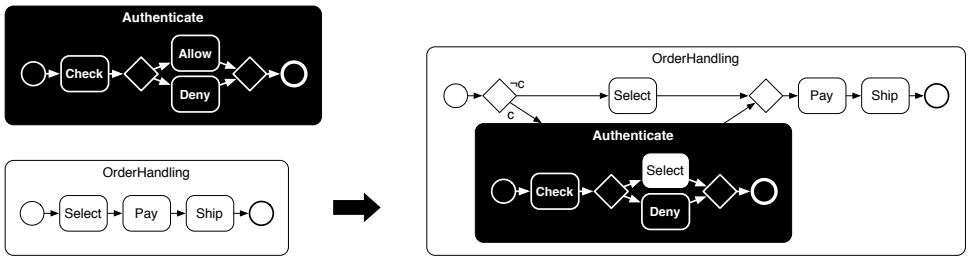



```
AuthenticateConnector:
CONNECT Authenticate
AROUND activity("OrderHandling\.Select")
PROCEEDING AT activity("Authenticate\.Allow")
```

Figure 4.22: Example around connector

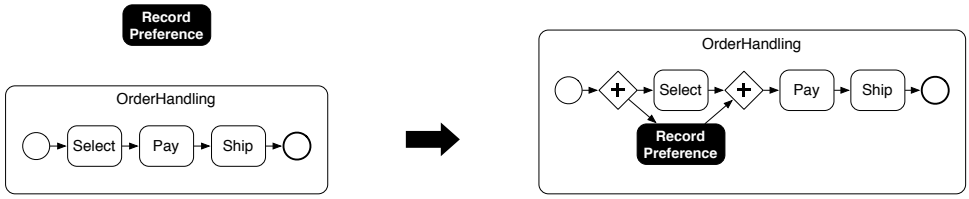
Around Connectors AroundConnectors specify that the advice Activity should be introduced into the base workflow *around* each of the joinpoints specified by the joinpointFragmentPointcut. Thus, these connectors implement the "around" concern connection pattern (CCP-4; cf. Section 4.4.3.4). For example, the around connector in Figure 4.22, which is named `AuthenticateConnector`, specifies that the `Authenticate` activity should be executed around the `Select` activity within the `OrderHandling` base workflow. The relative position of the joinpoint activity or fragment within the advice activity is specified using the *proceeding* clause, which consists of the `PROCEEDING AT` keywords followed by an activity or fragment pointcut.

In order to facilitate expressing combinations of the "around" pattern with other patterns, we offer a conditional variant of this pattern. For example, in Figure 4.23, the connector named `AuthenticateUnsecuredConnector` specifies that the `Authenticate` activity should be executed around the `Select` activity within the `OrderHandling` base workflow *if the current connection is unsecured.*




```
AuthenticateUnsecuredConnector:
CONNECT Authenticate
AROUND activity("OrderHandling\.Select")
IF "$connection.isUnsecured"
PROCEEDING AT activity("Authenticate\.Allow")
```

Figure 4.23: Example conditional around connector




```
RecordPrefsConnector:
CONNECT RecordPreference
PARALLEL TO activity("OrderHandling\.Select")
```

Figure 4.24: Example parallel connector

Parallel Connectors ParallelConnectors specify that the advice Activity should be introduced into the base workflow *in parallel with* each of the joinpoints specified by the joinpointFragmentPointcut. Thus, these connectors implement the "parallel" concern connection pattern (CCP-5; cf. Section 4.4.3.5). For example, the parallel connector in Figure 4.24, which is named `RecordPrefsConnector`, specifies that the `RecordPreference` activity should be executed in parallel with the `Select` activity within the `OrderHandling` base workflow.

In order to facilitate expressing combinations of the "parallel" pattern with other patterns, we offer a conditional variant of this pattern. For example, in Figure 4.25, the connector named `RecordPrefsIfOptedInConnector` specifies that the `RecordPreference` activity should be executed in parallel with the `Select` activity within the `OrderHandling` base workflow *if the customer has opted in to the preferences recording scheme.*

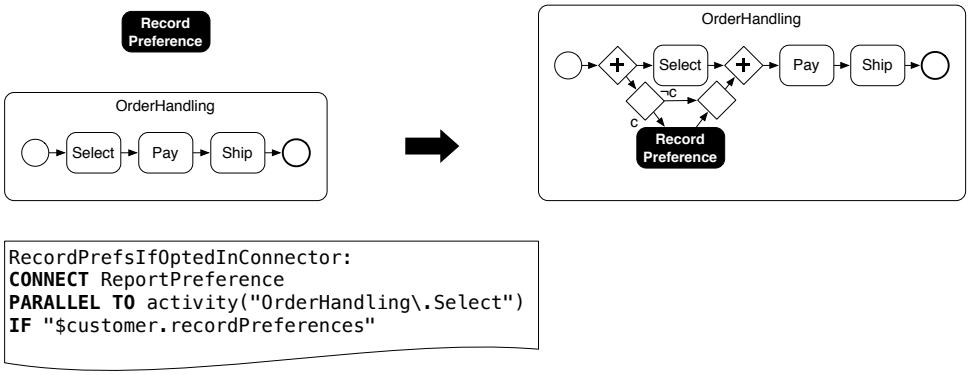


Figure 4.25: Example conditional parallel connector

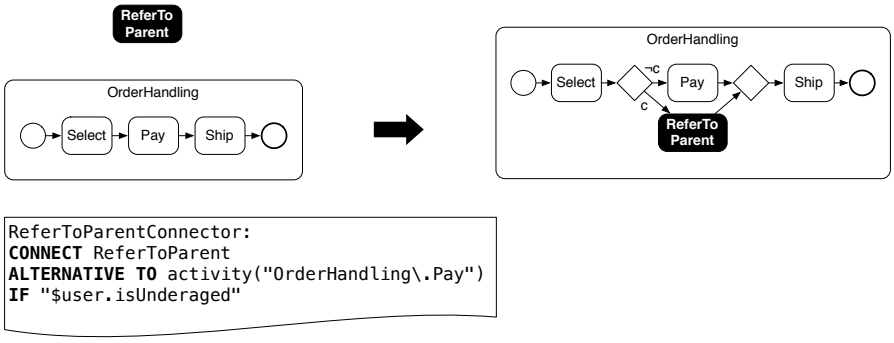


Figure 4.26: Example alternative connector

Alternative Connectors AlternativeConnectors specify that the advice Activity should be introduced into the base workflow *as an alternative to* each of the joinpoints specified by the joinpointFragmentPointcut. Thus, these connectors implement the "alternative" concern connection pattern (CCP-6; cf. Section 4.4.3.6). For example, the alternative connector in Figure 4.26, which is named `ReferToParentConnector`, specifies that the `ReferToParent` activity should be executed as an alternative to the `Pay` activity within the `OrderHandling` base workflow if the user is underaged.

Iterating Connectors IteratingConnectors specify that the advice Activity should be introduced into the base workflow *in an iteration with* each of the joinpoints specified by the joinpointFragmentPointcut. Thus, these connectors implement the "iterating" concern connection pattern (CCP-7; cf. Section 4.4.3.7). For example, the iterating connector in Figure 4.27, which is named `LoginFailureConnector`, specifies that the `HandleFailure` activity should be executed in an iteration with the `Login` activity within the `OrderHandling` base workflow until login succeeds.

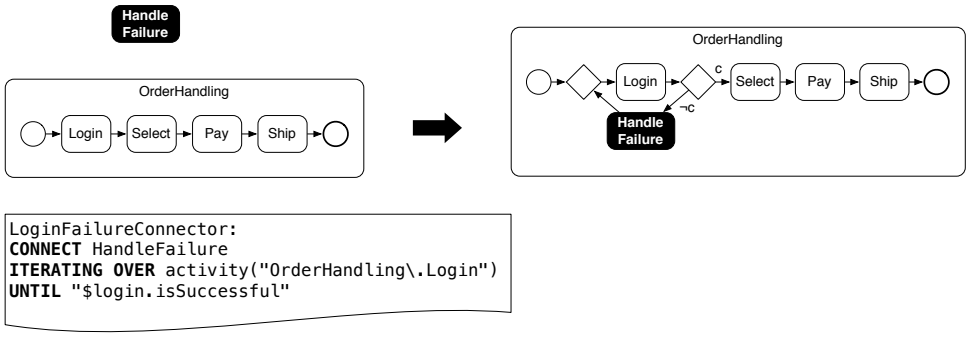


Figure 4.27: Example iterating connector

4.5.3.2 Internal Connectors

InternalConnectors implement the two internal concern connection patterns of Section 4.4.4. Using a one-to-one mapping from concern connection patterns to connectors, we obtain the synchronizing and switching connectors:

Synchronizing Connectors SynchronizingConnectors are used to specify that two branches of a parallel control structure should be *synchronized* through an advice activity. SynchronizingConnectors have three joinpoints. The first joinpoint is the parallel control structure in which the synchronization will occur, and is selected by the joinpointFragmentPointcut. The second joinpoint is the fragment in the first branch after which the synchronization will occur, and is selected by the splittingFragmentPointcut. The third joinpoint is the fragment in the second branch before which the synchronization will occur, and is selected by the joiningFragmentPointcut. Thus, these connectors implement the "synchronizing parallel branches" concern connection pattern (CCP-8; cf. Section 4.4.4.1).

For example, the synchronizing connector in Figure 4.28, which is named `VerifyAccountConnector`, specifies that the `VerifyAccount` activity should synchronize two branches of the parallel control structure that starts at the `OrderHandling` base workflow's `Split` node and ends at its `Join` node. The synchronization should occur after the `Pay` activity in one branch and before the `Ship` activity in another branch. Synchronizing these joinpoints implies that the control flow of the latter branch is blocked at its joinpoint until the control flow of the former branch has reached its joinpoint. This is accomplished by inserting a new AND-split after the former joinpoint and a new AND-join before the latter joinpoint, with the advice Activity in between.

Switching Connectors SwitchingConnectors are used to specify that two branches of a conditional control structure should be *switched* through an advice activity. SwitchingConnectors have three joinpoints. The first joinpoint is the conditional control structure in which the switch will occur, and is selected by the joinpointFragmentPointcut. The

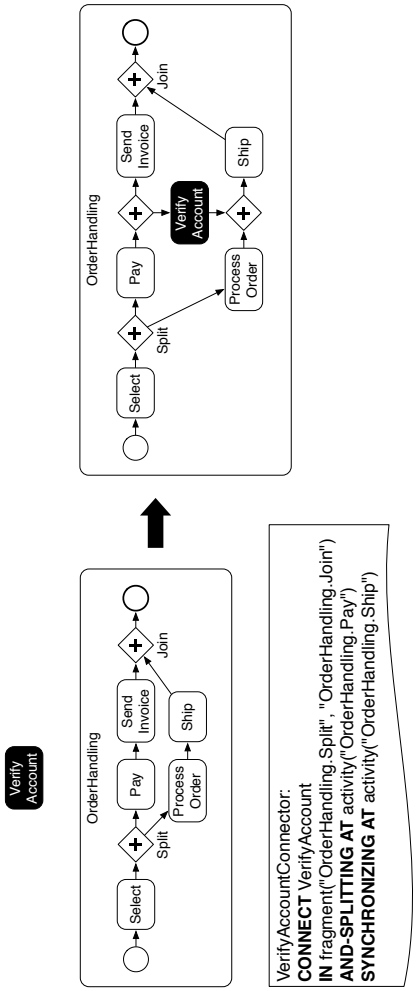


Figure 4.28: Example synchronizing connector

second joinpoint is the fragment in the first branch after which the switch will occur, and is selected by the splittingFragmentPointcut. The third joinpoint is the fragment in the second branch before which the switch will occur, and is selected by the joiningFragmentPointcut. Thus, these connectors implement the "switching alternative branches" concern connection pattern (CCP-9; cf. Section 4.4.4.2).

For example, the switching connector in Figure 4.29, which is named `CancelShipmentConnector`, specifies that the `CancelShipment` activity should switch two branches of the conditional control structure that starts at the `OrderHandling` base workflow's `Split` node and ends at its `Join` node. The switch should occur after the `PrepareShipment` activity in one branch and before the `HandOver` activity in another branch. Switching these joinpoints implies that the control flow of the former branch may need to be redirected to the latter branch from/to their respective joinpoints. This is accomplished by inserting a new XOR-split after the former joinpoint and a new XOR-join before the latter joinpoint, with the advice Activity in between.

This concludes our description of the different connectors offered by UNIFY. The complete syntax of the UNIFY connector language is specified in Backus–Naur form in Appendix A.

4.5.4 Aspect Composition Model

A connector represents the composition of one concern with another concern. Because more than one concern can be applicable to the same base workflow, the composition of more than two concerns should be supported in a way that prevents undesirable interactions between concerns. When analyzing these interactions, we observe that the joinpoints of a connector determine where possible interactions may occur: because all of our connectors give rise to the insertion of behavior directly around their joinpoints (for ExternalConnectors) or somewhere inside their joinpoints (for InsideConnectors), connectors can only interact with each other when their joinpoints intersect. Note that, because UNIFY uses Fragments as joinpoints, two connectors' joinpoints may be exactly the same, one connector's joinpoint may be completely contained in the other's, or two connectors' joinpoints may have a group of Nodes in common without one connector's joinpoint being completely contained in the other's.

We consider two main kinds of undesirable interactions. Firstly, when different connectors are applicable to the same joinpoint, the execution of the woven workflow must be deterministic, i.e., the relative ordering of the connectors' advices must be the same for each execution of the composition. Secondly, when different connectors are applicable to the same joinpoint, the application of one connector must not prevent the application of another connector or cancel the effects of another connector. In this section, we will outline a strategy for dealing with both kinds of undesirable interactions.

In general, UNIFY supports the application of any number of connectors within a workflow composition. However, for the purposes of investigating undesirable interactions, we will focus on interactions between only *two* connectors. A composition containing n connectors can then be analyzed by considering each of the $n(n-1)/2$ possible pairs of connectors in this composition. Table 4.6 lists possible undesirable interactions

Figure 4.29: Example switching connector

	Before	**After**	**Replace**	**Around**	**Parallel**	**Alt.**	**Iter.**
Before	1	(No int.)	2	1	3	4	5
After		1	2	1	3	4	5
Replace			2	2	2	2	2
Around				1, 6	3, 6	4, 6	5, 6
Parallel					3, 6	3, 4, 6	3, 5, 6
Alt.						4, 6	4, 5, 6
Iter.							5, 6

Table 4.6: Overview of interactions between external connectors; the numbers refer to the different kinds of interactions listed in Section 4.5.4. Because the complete table is symmetric around its main diagonal, we only show its upper right half.

between two ExternalConnectors that are applicable to the same joinpoint. These interactions are discussed below.

Interaction 1. *A composition with two BeforeConnectors or with a BeforeConnector and an AroundConnector is problematic if the entry nodes of the two connectors' joinpoint fragments are the same. In this situation, the order in which the connectors are applied determines the order in which the connectors' advices will be executed. Similarly, a composition with two AfterConnectors or with an AfterConnector and an AroundConnector is problematic if the exit nodes of the two connectors' joinpoint fragments are the same, and a composition with two AroundConnectors is problematic if the entry or exit nodes of the two connectors' joinpoint fragments are the same. A composition with two AroundConnectors that introduce a control structure around their joinpoints is also problematic if the two connectors' joinpoints have a group of Nodes in common without one connector's joinpoint being completely contained in the other's (cf. Interaction 6).*

Interaction 2. *A composition with a ReplaceConnector and another ExternalConnector is always problematic if both are applicable to the same joinpoint: if the ReplaceConnector is applied first, the joinpoint to which the other ExternalConnector is intended to be applied may no longer exist at the time of application. If the ReplaceConnector is applied last, the advice of the other ExternalConnector may be replaced.*

Interaction 3. *A composition with a ParallelConnector and a Before-, Around-, Alternative- or IteratingConnector is problematic if the entry nodes of the two connectors' joinpoint fragments are the same. In this situation, the order in which the connectors are applied determines whether the connectors' advices will be executed sequentially or in parallel to each other. Similarly, a composition with an After-, Around-, Alternative- or IteratingConnector is problematic if the exit nodes of the two connectors' joinpoint fragments are the same. A composition with two ParallelConnectors is only problematic if the two connectors' joinpoints have a group of Nodes in common without one connector's joinpoint being completely contained in the other's (cf. Interaction 6).*

Interaction 4. *A composition with an AlternativeConnector and a Before-, Around-, Parallel- or IteratingConnector is problematic if the entry nodes of the two connectors' joinpoint fragments are the same. In this situation, the order in which the connectors are*

applied determines whether the connectors' advices will be executed sequentially or as an alternative to each other. Similarly, a composition with an After-, Around-, Parallel- or IteratingConnector is problematic if the exit nodes of the two connectors' joinpoint fragments are the same. A composition with two AlternativeConnectors is only problematic if the two connectors' joinpoints have a group of Nodes in common without one connector's joinpoint being completely contained in the other's (cf. Interaction 6).

Interaction 5. *A composition with an IteratingConnector and a Before-, Around-, Parallel- or AlternativeConnector is problematic if the entry nodes of the two connectors' joinpoint fragments are the same. In this situation, the order in which the connectors are applied determines whether the non-IteratingConnector's advice will be part of the IteratingConnector's iteration or not. Similarly, a composition with an After-, Around-, Parallel- or AlternativeConnector is problematic if the exit nodes of the two connectors' joinpoint fragments are the same. A composition with two IteratingConnectors is only problematic if the two connectors' joinpoints have a group of Nodes in common without one connector's joinpoint being completely contained in the other's (cf. Interaction 6).*

Interaction 6. *A composition with two ExternalConnectors that introduce a control structure around their joinpoints (i.e., AroundConnectors that introduce a control structure around their joinpoints, ParallelConnectors, AlternativeConnectors and/or IteratingConnectors) is always problematic if the two connectors' joinpoints have a group of Nodes in common without one connector's joinpoint being completely contained in the other's. In this situation, the connector that is applied first will introduce an additional control flow path around its joinpoint. Because the connector that is applied second selects a joinpoint that starts or ends in the first connector's joinpoint, the first connector will have introduced additional incoming or outgoing edges into the fragment that is selected by the second connector, and the group of nodes selected by the second connector's pointcut will thus no longer be a fragment by the time the second connector is applied.*

Interactions 1, 3, 4 and 5 are related to the relative ordering of advices with respect to connectors' common joinpoints within a composition. Nondeterminism in the execution of the woven workflow can be prevented by requiring the workflow developer to specify the ordering in which the composition's connectors should be applied. Therefore, UNIFY introduces an explicit deployment construct named Composition. Similar to PADUS deployment specifications, UNIFY Compositions specify which process definition will act as the base workflow (cf. the <BaseConcern> element in Listing 4.1), and which behavior is applied to it. However, in PADUS, both the crosscutting behavior and its connection logic were encapsulated in the same aspect. In UNIFY, the behavior is specified as a regular CompositeActivity (cf. the <Concern> elements in Listing 4.1, which refer to BPEL process definitions in this example), while the connection logic is specified in a separate Connector (cf. the <Connector> elements in Listing 4.1). The order in which the connectors are specified determines the order in which they will be applied to the base workflow.

Interactions 2 and 6 are related to situations in which one connector may prevent the application of another connector. This is addressed in UNIFY by generating warnings during the weaving process when such interactions are detected. Further guidance

```
1 <Composition name="OrderHandling" separatorChar="/">
2   <BaseConcern>../concerns/OrderHandling/OrderHandling.bpel</BaseConcern>
3   <Concern>../concerns/Select/Select.bpel</Concern>
4   <Concern>../concerns/Report/Report.bpel</Concern>
5   <Concern>../concerns/RecordPreference/RecordPreference.bpel</Concern>
6   <Concern>../concerns/VerifyBankAccount/VerifyBankAccount.bpel</Concern>
7   <Connector>Select.connector</Connector>
8   <Connector>Report.connector</Connector>
9   <Connector>RecordPrefs.connector</Connector>
10  <Connector>VerifyBankAccount.connector</Connector>
11 </Composition>
```

Listing 4.1: An example composition consisting of a base concern and four other concerns that are applied to the base concern using four connectors

of the workflow developer in resolving such interactions is a field of research on itself and is therefore subject to future work. Note that we did not yet consider interaction issues of internal connectors. However, the above discussion can be extended towards internal connectors by considering their splitting joinpoint fragments as targets of an after advice, and their joining joinpoint fragments as targets of a before advice.

4.6 Discussion

UNIFY as a composition system The discipline of software engineering has originated in order to address the *software crisis* (Dijkstra, 1972) — the observation made in the late 1960s that the rapidly increasing size and complexity of software systems caused the development of these systems using existing approaches to become unmanageable. Over time, this has given rise to the construction of software by assembling existing (possibly *off-the-shelf*) *components* rather than rewriting every new software application from scratch. According to Aßmann (2003), component-based systems can be evaluated in terms of three main requirements:

- The **component model** determines how components appear: how are components defined, and what are their interfaces.
- The **composition technique** determines how components are composed: which *composition operators* (or *composers*) can be used.
- The **composition language** determines how a software system can be built by composing a number of components according to a certain *composition specification.*

When all three requirements are satisfied, one obtains a *composition system.* It should be clear that UNIFY can be considered a composition system: its components are the individual workflow concerns specified using the base language of Section 4.3. Its composition operators are the different connector types offered by the connector mechanism

as described in Section 4.5.3. Composition specifications take the form of UNIFY's connectors (cf. Section 4.5.3) and compositions (cf. Section 4.5.4).

A distinction can be made between composition systems composing black-box components and composition systems composing gray-box components. The latter can be divided in three main categories: aspect systems (such as ASPECTJ), control-flow based composition systems (such as *composition filters*; cf. Bergmans and Aksit, 2001), and composition expression systems (such as HYPERJ). *Invasive software composition* (Aßmann, 2003) is a composition approach that introduces a number of concepts allowing other composition approaches to be modeled as different variants of invasive software composition techniques. The main concepts are the following:

- A *fragment box* is a set of program elements, which has a composition interface that consists of a set of hooks. Thus, fragment boxes fulfill the role of components in invasive software composition.
- A *hook* is a point of variability of a fragment box, a set of fragment boxes, or a set of positions within fragment boxes subject to change. Hooks can be either *declared* or *implicit*.
- A *composer* is a program transformer that transforms one or more hooks for a reuse context.

It has already been shown that aspect systems can be considered a variant of invasive software composition (cf. Aßmann, 2003, Chapter 10). One can perform a similar exercise for UNIFY. CompositeActivities can be considered fragment boxes, every UNIFY Fragment constitutes an implicit hook, and the weaving of each Connector type constitutes a different composer. Nevertheless, integrating recent developments in composition operators and languages into UNIFY is subject to future work. For this purpose, recent work in extending invasive software composition to graphs (Johannes, 2010) and providing first-class composition operators for both aspectual and non-aspectual compositions (Havinga et al., 2010) seems promising.

4.7 Summary

UNIFY goes beyond the scope of PADUS by addressing a wider set of requirements, which are listed at the beginning of this chapter (cf. Section 4.1). In this chapter, we focus on a subset of these requirements:

1. We provide a uniform modularization mechanism that allows specifying both regular and crosscutting concerns using the same language construct, i.e., the CompositeActivity (Requirement 1).
2. We propose a coherent collection of seven external concern connection patterns and two internal concern connection patterns that recognize the specific characteristics of workflows, including the workflow paradigm's heavy focus on parallelism and choice (Requirement 2).

3. We provide a connector mechanism that allows independently modularized (regular and/or crosscutting) concerns to be connected according to the above concern connection patterns (Requirement 3).

   External connectors allow introducing behavior sequentially before, after, or around joinpoints, in parallel with joinpoints, as an alternative to joinpoints, or in iterations with joinpoints, while the workflow in which these joinpoints are located is oblivious of the connectors that may be applied to it. Thus, these connectors allow augmenting a concern with other concerns that were not considered when it was designed, which facilitates independent evolution and reuse of these concerns. The joinpoints are not limited to single activities, but can also be groups of workflow nodes that form a single-entry single exit (SESE) fragment. In addition to the above connectors that are mainly influenced by aspect-oriented principles, the *replace connector* allows expressing that an existing activity in one concern should be implemented by executing another concern, in a way that minimizes dependencies between these concerns and thus facilitates their independent evolution and reuse. Thus, this connector is related to traditional component-based software development (CBSD).

   Next to these external connectors, *internal connectors* allow introducing additional control flow dependencies within parallel or conditional control structures in order to allow synchronizing parallel branches or switching alternative branches.

4. At the heart of the approach lies a base language meta-model that allows expressing arbitrary workflows, and which can be instantiated towards several concrete workflow languages (Requirement 6).

In the following chapters, we focus on our other requirements: we provide our execution semantics (Requirement 4), build support for concern-specific abstractions (Requirement 5), and show how the framework's implementation is applicable to several concrete workflow languages and independent of a dedicated workflow engine (Requirement 7).

Chapter 5

A Formal Semantics for Aspect-Oriented Workflow Languages

In this chapter, we provide a formal semantics for the aspect-oriented workflow concepts that were introduced in the previous chapters. This will allow us to reason both on static and operational properties of workflows. Additionally, the formalization presented in this chapter supports the implementation of the UNIFY *framework, which is presented in Chapter 7. Readers who are interested more in* UNIFY*'s complete functionality than in its formalization may want to read Chapter 6 and onwards first, and come back to this chapter later.*

5.1 Motivation and Requirements

Because workflow management systems and business process management systems (cf. Section 2.1.4.1) are driven by workflows, the correctness of workflows is essential to these systems. Unfortunately, commercial systems typically do not support the verification of workflows (van der Aalst et al., 2011). Moreover, as is shown in various case studies (e.g., Mendling et al., 2007), process designers tend to make many errors. Typical errors are deadlocks (a case gets stuck), livelocks (a case cannot progress), and other anomalies. Repairing such errors can be time consuming and costly. Therefore, workflow verification is highly relevant (van der Aalst et al., 2011).

The topic of workflow verification has been researched for more than a decade. van der Aalst (1997, 1998a, 2000) have proposed the notion of *workflow nets* (WF-nets), which constitute a mapping of workflow management concepts to Petri nets (Petri and Reisig, 2008), and thus introduce, among others, notions of process definitions, routing constructs, and activities. Their approach allows verifying the property of *soundness* of WF-nets. *Classical soundness* (van der Aalst, 1997, 1998a) is defined in terms of a workflow having (1) the option to complete, (2) proper completion, and (3) no dead transitions (van der Aalst et al., 2011).

van der Aalst (1998b) proposes three reasons for using Petri nets: (1) they have a formal semantics in addition to their graphical nature, (2) they are state-based instead of (just) event-based, and (3) there is an abundance of analysis techniques for Petri nets. Indeed, approaches based on Petri nets have become a popular means of formalizing workflows and subsequently verifying properties such as soundness. For example, Ouyang et al. (2007) and Lohmann (2007) have each formalized the execution semantics of WS-BPEL using Petri nets, and Dijkman et al. (2008) have formalized the execution semantics of BPMN. YAWL has been developed by augmenting high-level Petri nets with additional constructs (thus obtaining *extended workflow nets*; cf. van der Aalst and ter Hofstede, 2005).

Nevertheless, Petri nets aren't the only formalization suited to our approach. In particular, when reviewing our base language meta-model as presented in Section 4.3, we can consider a UNIFY workflow to be a graph consisting of certain types of nodes, each of which may have certain attributes. We can then consider the application of a UNIFY connector to be a transformation of such a graph. All of this maps naturally to the Graph Transformation formalism (Rozenberg, 1997; Ehrig et al., 2006), which allows defining a graph transformation system consisting of a *type graph* specifying the types and attributes of possible graph nodes and edges, as well as a number of *graph transformation rules* specifying how *typed attributed graphs* conforming to the type graph can be transformed into another typed attributed graph. The formalism enables static verification of these transformation rules based on *critical pairs analysis*, which allows the detection of possible mutual exclusions and causal dependencies between transformation rules.

Because UNIFY's connector mechanism constitutes a novel workflow modularization mechanism, we must ensure that the connector mechanism's semantics is precisely described, and that this semantics fits into the workflow community's existing formal tradition. Therefore, we aim to provide a formalization of our approach that is compatible with existing research on this topic within the workflow community, but also addresses the specific notion of connection patterns introduced by UNIFY.

5.2 Towards a Formalization of Aspect-Oriented Workflow Languages

The goal of the work described in this chapter is twofold: on the one hand, we want to be able to verify certain properties of workflows, which can be either static (e.g., applicability and effects of connectors) or dynamic (e.g., absence of possible deadlocks or livelocks), while on the other hand, we want to provide a basis for implementing aspect-oriented workflow languages.

In order to formalize the aspect-oriented workflow concepts introduced in the previous chapters, we employ two complementary formalisms. First, we augment the static description of UNIFY's workflows as provided by its base language and connector language meta-models with a static semantics for the weaving of UNIFY connectors using the Graph Transformation formalism. This facilitates static reasoning over the applicability and effects of connectors, and can be used to implement a static weaver of UNIFY connectors. Second, we provide a semantics for the operational properties of workflows by proposing a translation to Petri nets, and subsequently extend this semantics to sup-

Part of the approach	Use for analysis	Use for implementation
Graph Transformation formalization of connectors (cf. Section 5.3)	Applicability and effects of workflow connectors	Static weaver
Petri net formalization of concerns (cf. Section 5.4.2)	Dynamics of workflow concerns	Dedicated workflow engine
Petri net formalization of connectors (cf. Section 5.4.3)	Dynamics of workflow connectors	Dynamic weaver

Table 5.1: Overview of the approach

port the operational effects of connectors. This allows reasoning on the dynamics of UNIFY workflow compositions, and can be used to implement a dedicated workflow engine for UNIFY. Table 5.1 gives an overview of our approach.

5.3 Graph Transformation Formalization of Connectors

5.3.1 The Graph Transformation Formalism

The semantics of a connector, which connects an advice concern to a base concern, is given by constructing a new concern that composes the base concern and the advice concern according to the connector type and the pointcut specification. This is accomplished using *graph transformation rules* that work on the abstract syntax of the UNIFY base language.

A *graph* consists of a set of nodes and a set of edges. A *typed graph* is a graph in which each node and edge belong to a type defined in a *type graph*. An *attributed graph* is a graph in which each node and edge may contain attributes where each attribute is a *(value, type)* pair giving the value of the attribute and its type. Types can be structured by inheritance relations.

A *graph transformation rule* is a rule used to modify a host graph G, and is defined by two graphs (L, R). L is the left-hand side of the rule representing the pre-conditions of the rule and R is the right-hand side representing the post-conditions of the rule. Note that the left-hand side of the rule can be composed of a positive application condition (presence of certain combinations of nodes and edges) and a set of negative application conditions or NACs (absence of certain combinations of nodes and edges). The process of applying the rule to a graph G involves finding a graph monomorphism $h : L \mapsto G$ and replacing $h(L)$ in G with $h(R)$. Further details can be found in (Rozenberg, 1997).

In our approach, the type graph represents the UNIFY base language meta-model that was already presented earlier in Figure 4.2. The translation of this meta-model to a type graph is straightforward: each meta-class corresponds to a typed node and each meta-association corresponds to a typed edge. Attributes in the meta-model are translated to corresponding node attributes. The wellformedness constraints can be formalized by graph constraints. Figure 5.1 shows a screenshot of UNIFY's type graph and `Before` rule in the state-of-the-art Graph Transformation analysis tool AGG (Taentzer et al., 2009).

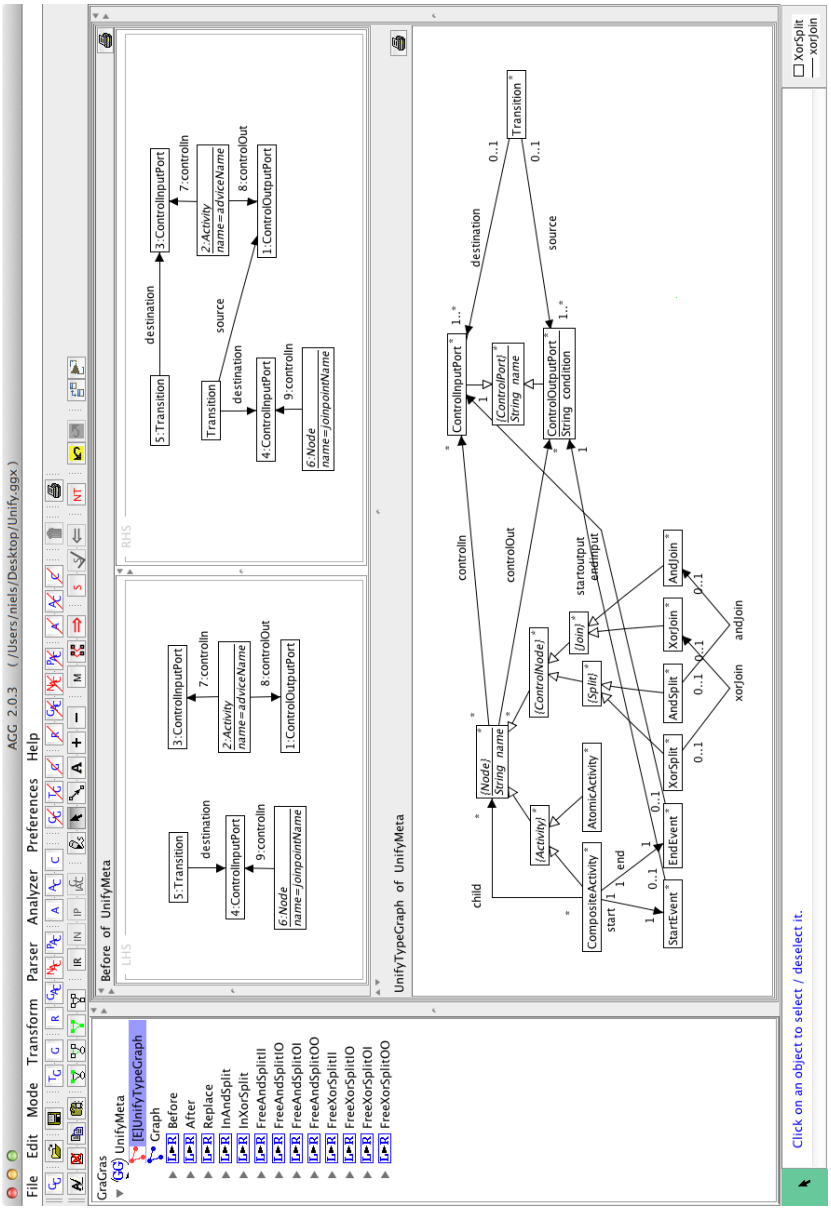


Figure 5.1: Screenshot of UNIFY's type graph (bottom) and Before rule (top) in AGG

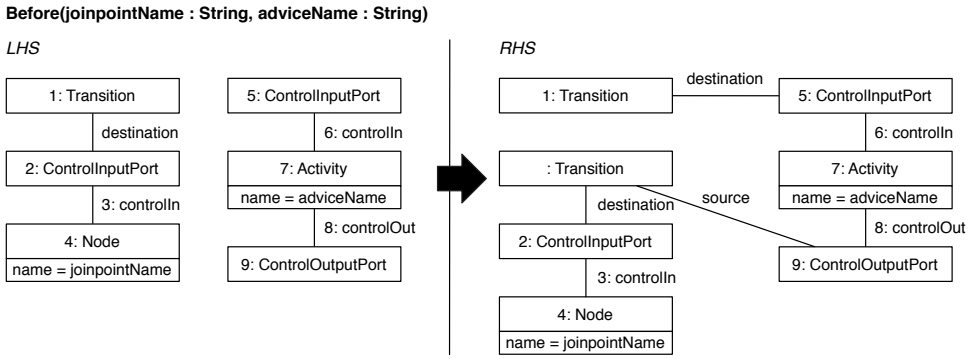


Figure 5.2: The `Before` graph transformation rule

5.3.2 Graph Transformation Rules

Each UNIFY connector's weaving semantics is specified by a graph transformation rule, which is parametrized by the name of a joinpoint Activity and the name of the advice Activity. Thus, a specific UNIFY connector application gives rise to the application of a graph transformation rule for each of the joinpoint Activities selected by the connector's pointcut, with the advice being the same for each of these rule applications.

5.3.2.1 Before Connectors

The rule for the BeforeConnector is parametrized by the name of a joinpoint Activity, and the name of the advice Activity that should be added before it. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint Activity names. Each name is the input for a rule application. Figure 5.2 shows the **Before(joinpointName : String, adviceName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an Activity whose name is the value of the `joinpointName` parameter, together with its ControlInputPort and the Transition that is connected to it) and the advice Activity named `adviceName` with its input and output ports. The right-hand side of the rule shows the connection of the original Transition to the advice Activity's ControlInputPort, and of the advice Activity's ControlOutputPort to the joinpoint Activity's ControlInputPort through a new Transition.

5.3.2.2 After Connectors

The rule for the AfterConnector is similar to the rule for the BeforeConnector. It is parametrized by the name of a joinpoint Activity, and the name of the advice Activity that should be added after it. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint Activity names. Each name is the input

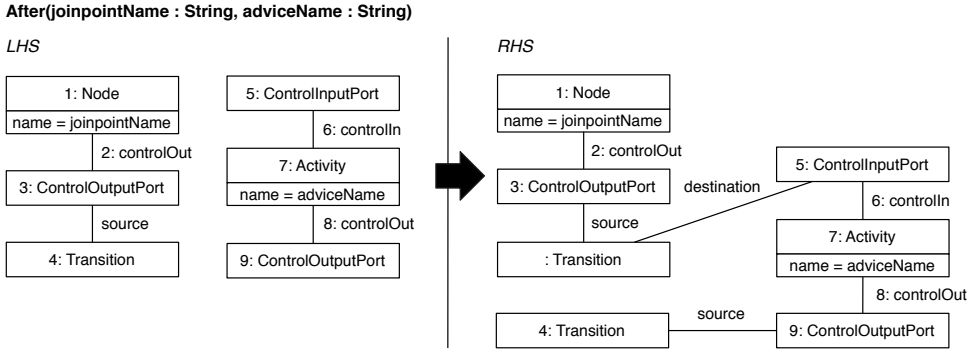


Figure 5.3: The `After` graph transformation rule

for a rule application. Figure 5.3 shows the **After(joinpointName : String, adviceName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an Activity whose name is the value of the `joinpointName` parameter, together with its ControlOutputPort and the Transition that is connected to it) and the advice Activity named `adviceName` with its input and output ports. The right-hand side of the rule shows the connection of the joinpoint Activity's ControlOutputPort to the advice Activity's ControlInputPort through a new Transition, and of the advice Activity's ControlOutputPort to the original Transition.

5.3.2.3 Replace Connectors

The rule for the ReplaceConnector is parametrized by the name of a joinpoint Activity, and the name of the advice Activity that should replace it. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint Activity names. Each name is the input for a rule application. Figure 5.4 shows the **Replace(joinpointName : String, adviceName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented: an Activity whose name is the value of the `joinpointName` parameter (together with its ControlInputPort and ControlOutputPort, and the Transitions that are connected to them), and the advice Activity named `adviceName` with its control input and output ports. The right-hand side of the rule shows the connection of the original incoming Transition to the advice Activity's ControlInputPort, and of the advice Activity's ControlOutputPort to the original outgoing Transition. The joinpoint Activity is thus removed from the graph.

5.3.2.4 Around Connectors

The rule for the AroundConnector is parametrized by the name of a joinpoint Activity, the name of the advice CompositeActivity that should be woven around it, and the name of the proceed Activity (which is a child of the advice CompositeActivity and in-

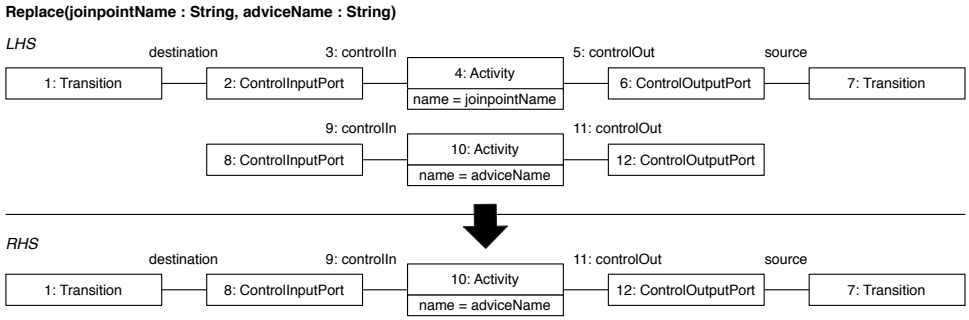


Figure 5.4: The `Replace` graph transformation rule

dicates where the joinpoint Activity should occur within the advice). The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint Activity names. Each name is the input for a rule application. Figure 5.5 shows the **Around(joinpointName : String, adviceName : String, proceedName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented: an Activity whose name is the value of the `joinpointName` parameter (together with its ControlInputPort and ControlOutputPort, and the Transitions that are connected to them), the advice CompositeActivity named `adviceName` with its control input and output ports, and the advice CompositeActivity's child Activity named `proceedName` (together with its ControlInputPort and ControlOutputPort, and the Transitions that are connected to them). The right-hand side of the rule shows the connection of the original incoming Transition to the advice Activity's ControlInputPort, and of the advice Activity's ControlOutputPort to the original outgoing Transition. As a child of the advice Activity, the proceed Activity is replaced by the joinpoint Activity.

5.3.2.5 Parallel Connectors

The rule for the ParallelConnector is parametrized by the name of a joinpoint Activity, and the name of the advice Activity that should be added parallel to it. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint Activity names. Each name is the input for a rule application. Figure 5.6 shows the **Parallel(joinpointName : String, adviceName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an Activity whose name is the value of the `joinpointName` parameter, together with its ControlInputPort and the incoming Transition that is connected to it, and its ControlOutputPort and the outgoing Transition that is connected to it) and the advice Activity named `adviceName` with its input and output ports. The right-hand side of the rule shows the connection of the original incoming Transition to a new AndSplit (through a new ControlInputPort). The new AndSplit has two outgoing branches: the first connects the new

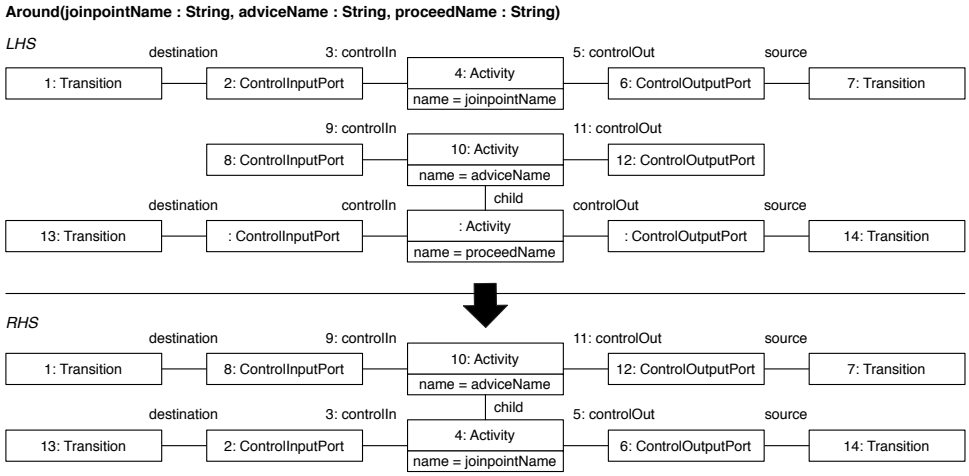


Figure 5.5: The Around graph transformation rule

AndSplit to the joinpoint Activity's ControlInputPort (through a new ControlOutputPort and Transition), while the second connects the new AndSplit to the advice Activity's ControlInputPort (through a new ControlOutputPort and Transition). The joinpoint Activity's ControlOutputPort is connected to a new AndJoin (through a new Transition and ControlInputPort), just like the advice Activity's ControlOutputPort is connected to this new AndJoin (through a new Transition and ControlInputPort). Finally, the new AndJoin is connected to the joinpoint Activity's original outgoing Transition (through a new ControlOutputPort).

5.3.2.6 Alternative Connectors

The rule for the AlternativeConnector is similar to the rule for the ParallelConnector. It is parametrized by the name of a joinpoint Activity, the name of the advice Activity that should be added alternative to it, and the condition that decides whether the alternative branch should be followed or not. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint Activity names. Each name is the input for a rule application. Figure 5.7 shows the **Alternative(joinpointName : String, adviceName : String, condition : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an Activity whose name is the value of the `joinpointName` parameter, together with its ControlInputPort and the incoming Transition that is connected to it, and its ControlOutputPort and the outgoing Transition that is connected to it) and the advice Activity named `adviceName` with its input and output ports. The right-hand side of the rule shows the connection of the original incoming Transition to a new XorSplit (through a new ControlInputPort). The new XorSplit has two outgoing branches: the first connects the new XorSplit to the joinpoint

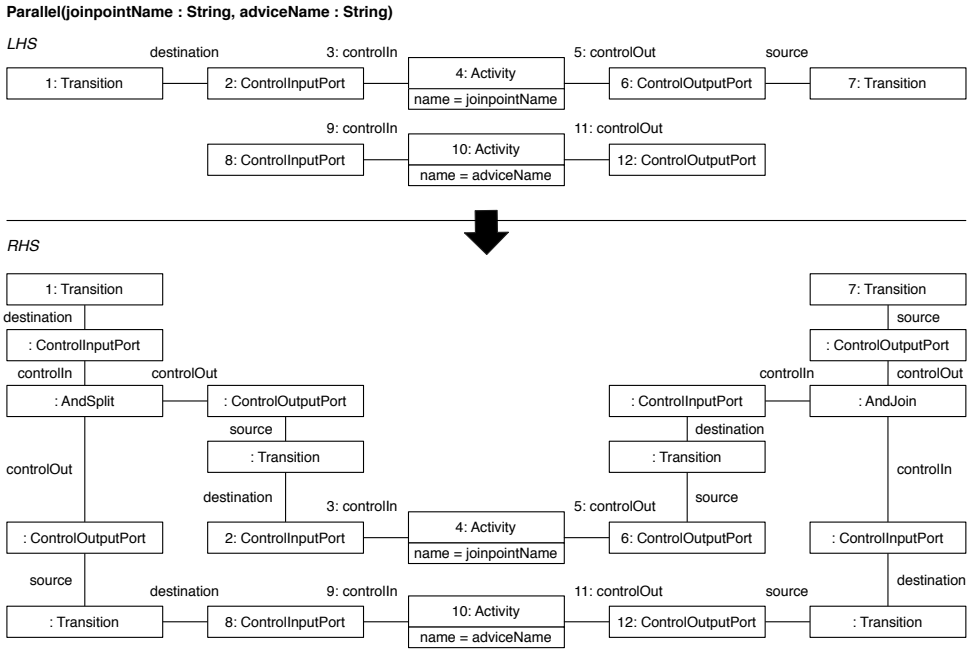


Figure 5.6: The `Parallel` graph transformation rule

Activity's ControlInputPort (through a new ControlOutputPort and Transition), while the second connects the new XorSplit to the advice Activity's ControlInputPort (through a new ControlOutputPort and Transition) using the specified condition. The joinpoint Activity's ControlOutputPort is connected to a new XorJoin (through a new Transition and ControlInputPort), just like the advice Activity's ControlOutputPort is connected to this new XorJoin (through a new Transition and ControlInputPort). Finally, the new XorJoin is connected to the joinpoint Activity's original outgoing Transition (through a new ControlOutputPort).

5.3.2.7 Iterating Connectors

The rule for the IteratingConnector is parametrized by the name of a joinpoint Activity, the name of the advice Activity that should be added in an iteration with it, and the condition that decides when the iteration should finish. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint Activity names. Each name is the input for a rule application. Figure 5.8 shows the **Iterating(joinpointName : String, adviceName : String, condition : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an Activity whose name is the value of the `joinpointName` parameter, together with its ControlInputPort and the incoming Transition that is connected to it, and its

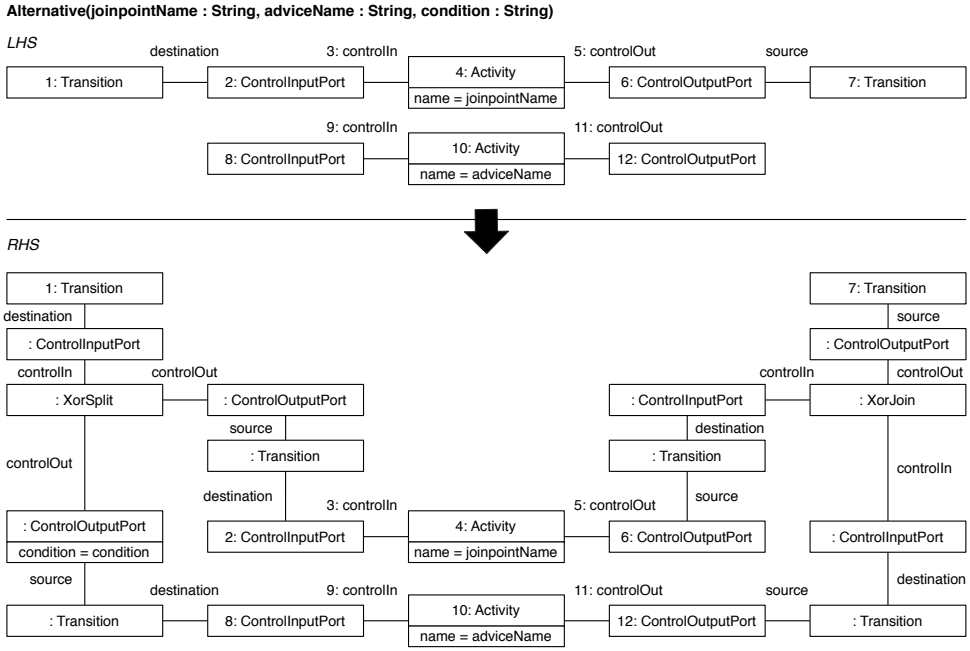


Figure 5.7: The `Alternative` graph transformation rule

ControlOutputPort and the outgoing Transition that is connected to it) and the advice Activity named `adviceName` with its input and output ports. The right-hand side of the rule shows the connection of the original incoming Transition to a new XorJoin (through a new ControlInputPort). The XorJoin is connected to the joinpoint Activity's ControlInputPort (through a new ControlOutputPort and Transition). The joinpoint Activity's ControlOutputPort is connected (through a new Transition and ControlInputPort) to a new XorSplit, which is connected to the original outgoing Transition (through a new ControlOutputPort, which ensures this branch of the XorSplit is only taken when the specified condition evaluates to true). The XorSplit has a second outgoing branch which connects (through a new ControlOutputPort and Transition) to the advice Activity's ControlInputPort. Finally, the advice Activity's ControlOutputPort is connected (through a new Transition and ControlInputPort) to the new XorJoin, thus completing a structured loop in the control flow represented by this graph.

5.3.2.8 Synchronizing Connectors

The rule for the SynchronizingConnector is parametrized by the name of a splitting joinpoint Activity, the name of a joining joinpoint Activity, and the name of the advice Activity that should be inserted between these two. Figure 5.9 shows the **Synchronizing(sJoinpointName : String, jJoinpointName : String, adviceName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be aug-

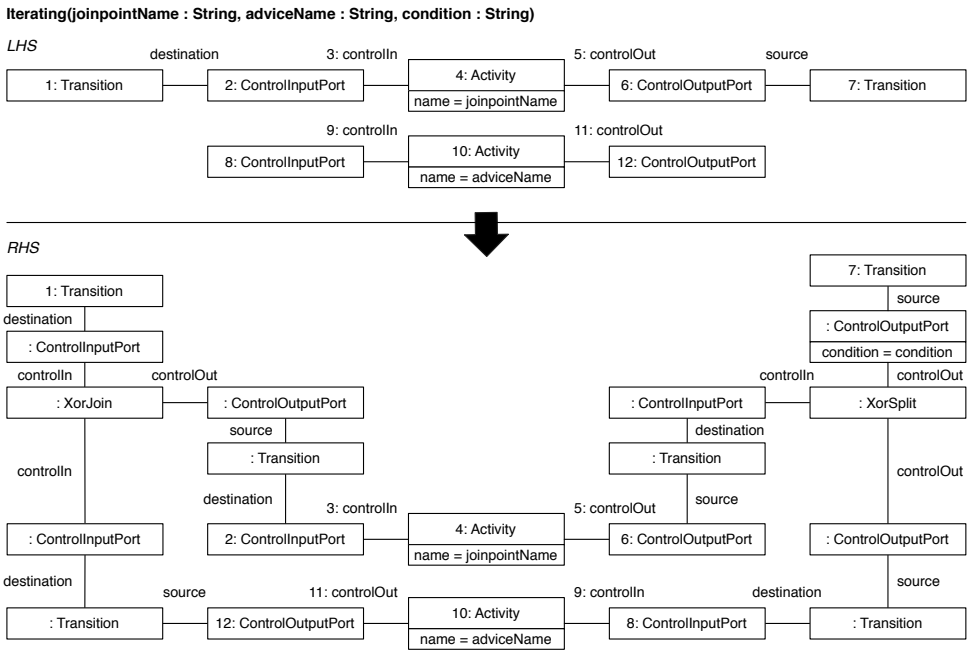


Figure 5.8: The `Iterating` graph transformation rule

mented (i.e., an Activity whose name is the value of the `sJoinpointName` parameter, together with its ControlOutputPort and the outgoing Transition that is connected to it, and an Activity whose name is the value of the `jJoinpointName` parameter, together with its ControlInputPort and the incoming Transition that is connected to it) and the advice Activity named `adviceName` with its input and output ports. The right-hand side of the rule shows the connection of the splitting joinpoint Activity to a new AndSplit through a new Transition. The AndSplit has one outgoing Transition which connects to the splitting joinpoint Activity's original outgoing Transition, and another outgoing Transition which connects to the advice Activity. The advice Activity is connected to a new AndJoin through a new Transition; the AndJoin's other incoming Transition is the joining joinpoint Activity's original incoming Transition. Finally, the AndJoin's outgoing Transition connects to the joining joinpoint Activity.

5.3.2.9 Switching Connectors

The **Switching(sJoinpointName : String, jJoinpointName : String, adviceName : String, condition : String)** rule for the SwitchingConnector, which is shown in Figure 5.10, is analogous to the rule for the SynchronizingConnector: the only difference is that it inserts an XorSplit (with the appropriate condition) and an XorJoin instead of an AndSplit and an AndJoin, respectively.

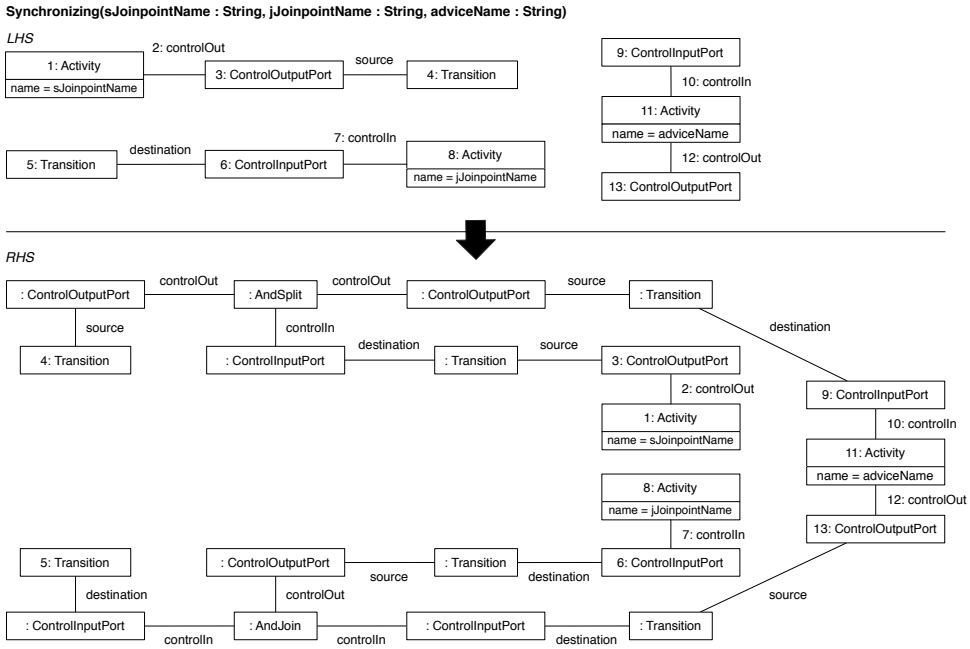


Figure 5.9: The Synchronizing graph transformation rule

5.3.3 Analysis

The graph transformation rules presented above constitute a precise specification for the weaving of an advice Activity into a base workflow according to a Connector. As we will discuss in Chapter 7, UNIFY's source code weaver is a manual JAVA implementation of the weaving semantics specified by each of these rules. However, the benefits of this formalization are not limited to the implementation of such static weaving: it can also be used to statically analyze the effects of UNIFY connectors. This analysis is based on the formal notion of *indepencence* of graph transformations, which expresses the fact that, in a given context, two transformations are neither mutually exclusive nor causally dependent. Thus, a distinction can be made between the notions of *parallel independence* (absence of mutual exclusions) and *sequential independence* (absence if causal dependencies). A formal treatment of these concepts is provided by Ehrig et al. (2004), while a good introduction to these concepts (and their application to the detection and resolution of model inconsistencies) is provided by Mens et al. (2006).

Based on the notion of independence, a potential *parallel* or *sequential dependency* is defined as a pair of transformation rules for which a counterexample to parallel or sequential independence can be found. More precisely, two rules are mutually exclusive if the application of the first prevents the application of the second, or vice versa. Two rules are causally dependent if the application of the second requires prior application of the first. *Critical pairs analysis* (Plump, 1993) can be used to compute all potential

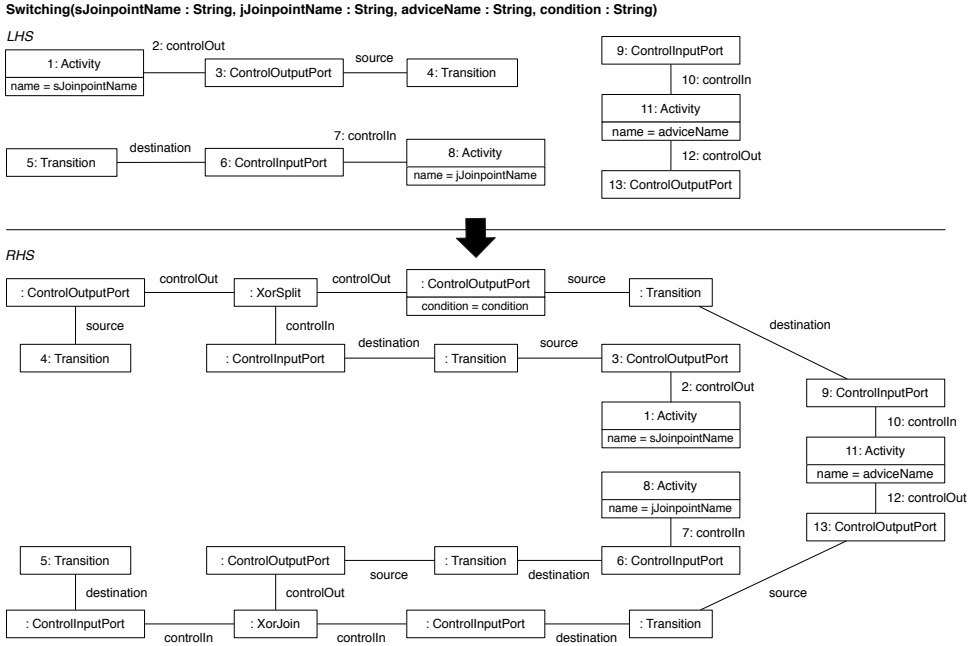


Figure 5.10: The `Switching` graph transformation rule

mutual exclusions and causal dependencies for a given set of transformation rules by pairwise comparison. Critical pairs formalize the idea of showing a conflicting situation in a minimal context. Critical pairs identifying a mutual exclusion can be computed by comparing the left-hand sides of two rules: a (partial) overlap between both indicates a potential mutual exclusion. Critical pairs identifying a causal dependency can be computed by comparing the right-hand side of one rule with the left-hand side of another: a (partial) overlap between both indicates a potential causal dependency (Mens et al., 2006). To perform such critical pairs analysis, we use the state-of-the-art Graph Transformation analysis tool AGG (Taentzer et al., 2009).

In our initial experiments applying critical pairs analysis to our graph transformation rules, we had some difficulty obtaining reasonable execution times for the computation of critical pairs. After conferring with the authors of AGG, we concluded that this was due to the fact that our rules are relatively large compared to the typical application domains of the Graph Transformation formalism (with large rules giving rise to an explosion of the space of possible overlapping graphs). Therefore, we have simplified the type graph and rules somewhat: (1) We have removed the ControlPort, ControlInputPort and ControlOutputPort nodes from the type graph, and replaced these by direct associa-

Minimal Conflicts

Show

first \ second	1	2	3	4	5	6	7	8	9
1 Before	2	0	1	1	1	1	1	1	1
2 After	0	2	1	1	1	1	1	1	1
3 Replace	1	1	2	1	1	1	1	0	0
4 Around	1	1	1	8	1	1	1	6	6
5 Parallel	1	1	1	1	2	1	1	2	2
6 Alternative	1	1	1	1	1	2	1	2	2
7 Iterating	1	1	1	1	1	1	2	2	2
8 Synchronizing	1	1	0	6	2	2	2	206	0
9 Switching	1	1	0	6	2	2	2	0	212

Figure 5.11: Numbers of mutual exclusions between graph transformation rules as computed by AGG

tions between a Node and its incoming and outgoing Transitions.[1] (2) We have removed the ControlNode node from the type graph, as this abstract node can be replaced by direct inheritance links between its children and its parent.[2] These changes caused the critical pairs analysis to finish within a more acceptable timeframe.[3] Note that the type graph and rules presented earlier in this chapter are the unsimplified ones. The final AGG formalization of UNIFY, as well as the complete analysis of its critical pairs, can be downloaded from the UNIFY website.[4]

Figure 5.11 shows the numbers of critical pairs identifying mutual exclusions as computed by AGG. Note that these critical pairs are *delete–use conflicts*, as they are caused by one rule deleting a part of a graph used in another rule's left-hand side. We have analyzed each of these conflicts. Summarizing this analysis, the conflicts include the following main cases:

- The conflicts of most rule combinations indicate situations where the application

[1] Nevertheless, control ports remain an integral part of the UNIFY base language meta-model as presented in Chapter 4. They are especially significant in view of the extensibility of the meta-model, e.g., towards the data and exception handling perspectives, which may benefit from the notion of *data* and/or *exception ports*.

[2] Nevertheless, control nodes are an important conceptual element of the UNIFY base language meta-model that clearly encapsulate the commonalities of splits and joins.

[3] Computing possible mutual exclusions takes about 15 minutes on an Intel Core i5-2500K processor with 2 GB of JAVA heap space. Computing possible causal dependencies takes several hours (due to the need to include the — larger — right-hand sides in the comparison of rules).

[4] Cf. `http://www.unify-framework.org/GraphTransformation.tgz`.

of one rule breaks the association between the joinpoint and its incoming and/or outgoing transition (because an advice is being woven before, after, or around the joinpoint). Thus, the second rule can no longer be applied, because its left-hand side no longer exists in the graph. This case is not necessarily problematic in UNIFY, because UNIFY pointcut expressions only refer to the joinpoint, and not to its incoming and/or outgoing transition. Nevertheless, it indicates that the ordering of connector applications is important, which confirms the observations we made in Section 4.5.4.

- The conflicts of rule combinations where the first rule is the `Replace` rule indicate situations where the application of the first rule removes the joinpoint of the second rule. This case is problematic in UNIFY, and is addressed in Section 4.5.4 by generating warnings during the weaving process when such interactions are detected.

- The many conflicts of the combination of two `Synchronizing` rules or the combination of two `Switching` rules are caused by the large size of these rules' left-hand sides, which gives rise to a large amount of overlaps between the rules' joinpoints. Nevertheless, these conflicts do not indicate problematic situations in UNIFY: in UNIFY, applying a synchronizing or switching connector requires the existence of a parallel or alternative control structure, respectively. The weaving of the connector breaks this control structure: because two of its branches are now connected, it is no longer a control structure but rather an arbitrary workflow fragment. Thus, it can no longer be the target of the application of a second synchronizing and switching connector, and is similar to the above case involving the replace connector: a warning can be generated during the weaving process when such an interaction is detected.

Figure 5.12 shows the numbers of critical pairs identifying causal dependencies as computed by AGG. Note that these critical pairs are *produce–use dependencies*, as they are caused by one rule producing a part of a graph used in another rule's left-hand side. We have analyzed these dependencies, and have identified the following main cases:

- Nearly all rule applications — except the `Replace` rule — introduce new parts into the graph, which enable the application of other rules. This is expected and not necessarily problematic in UNIFY, as it merely signifies that the ordering of connector applications is important, and thus again confirms the observations we made in Section 4.5.4.

- Many critical pairs of this analysis constitute graphs that do not represent valid UNIFY workflows, e.g., graphs where the advice and the joinpoint form a loop that is separate from the rest of the workflow, graphs where two rules' advice overlap, graphs where two fragments have the same incoming or outgoing transition.

- The combination of two `Synchronizing` rules or two `Switching` rules, and — to a lesser extent — the combination of a `Synchronizing` or `Switching` rule with an

Minimal Dependencies

Show

first \ second	1	2	3	4	5	6	7	8	9
1 Before	2	0	2	3	2	2	2	2	2
2 After	0	2	2	3	2	2	2	2	2
3 Replace	0	0	0	0	0	0	0	0	0
4 Around	1	1	0	6	1	1	1	6	6
5 Parallel	1	1	1	7	2	1	1	6	6
6 Alternative	1	1	1	7	1	2	1	6	6
7 Iterating	1	1	1	7	1	1	2	6	6
8 Synchronizing	1	1	0	26	6	6	6	646	0
9 Switching	1	1	0	26	6	6	6	0	390

Figure 5.12: Numbers of causal dependencies between graph transformation rules as computed by AGG

Around rule, lead to a large number of detected dependencies. Selectively analyzing a number of these, we see a lot of invalid graphs as described in the previous case. We also see some valid graphs where the application of the first rule enables the application of the second, as described in the first case. The much larger number of such dependencies is due to the large size of these rules.

We can thus conclude that the critical pairs analysis does not reveal any significant shortcomings to the UNIFY connector mechanism as presented in Chapter 4, as the critical pairs containing valid graphs indicate situations that are addressed by UNIFY's pointcut language (which is evaluated by a precise pointcut matching process based on the names of the joinpoint activities), advice model (which allows introducing behavior into a base workflow by weaving a copy of an advice activity) or composition model (which allows specifying the precedence strategy according to which connectors should be woven, and addresses cases where one connector application prevents the application of another connector). This confirms the observations we made in Section 4.5.4. Nevertheless, the Graph Transformation formalism does not seem perfectly well suited to some of our larger rules, most notably the ones for the synchronizing and switching connectors.

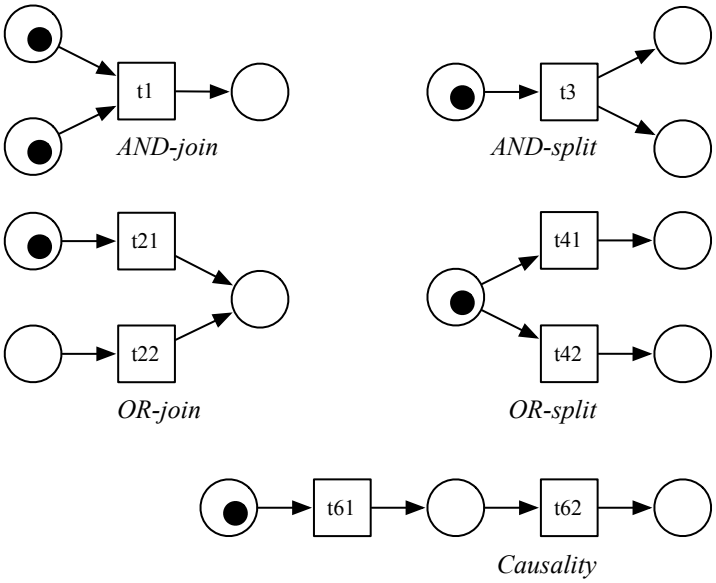


Figure 5.13: Petri net patterns for workflow primitives (van der Aalst, 1998b)

5.4 Petri Net Formalization of Concerns and Connectors

5.4.1 Existing Petri Net Formalizations of Workflows

The most accepted track of existing research into formalizing workflows using Petri nets is the research by van der Aalst *et al.* (van der Aalst, 1998a,b; van der Aalst and ter Hofstede, 2005), which has led to the development of workflow nets (WF-nets) and YAWL (Yet Another Workflow Language). Their formalization consists of introducing Petri net patterns for each of the workflow primitives, i.e., *AND-join*, *AND-split*, *OR-join*, *OR-split*, and *causality* (or *sequence*). These patterns are shown in Figure 5.13. In YAWL (van der Aalst and ter Hofstede, 2005), these primitives are augmented with additional primitives for specifying multiple instances of workflow activities, and for canceling workflow activities, but as UNIFY does not aim to support these, we will not consider these further.

We believe these patterns form an excellent basis for specifying a semantics for our own UNIFY base language primitives. However, when reviewing the approach further, we note the absence of a precise algorithm for applying these patterns to a workflow in order to generate its corresponding Petri net in a way that provides a clear mapping between workflow elements and their corresponding Petri net elements. For example, Figure 5.14 shows a workflow and its corresponding Petri net as provided by van der Aalst (1998b). In this example, it is not trivial to see how transitions D, E, and F, together with their input and output places, have been obtained in the corresponding Petri net: we would have expected something more along the lines of Figure 5.15, in which the application of the OR-join pattern to task F mirrors the application of the OR-split pattern to tasks B and C.

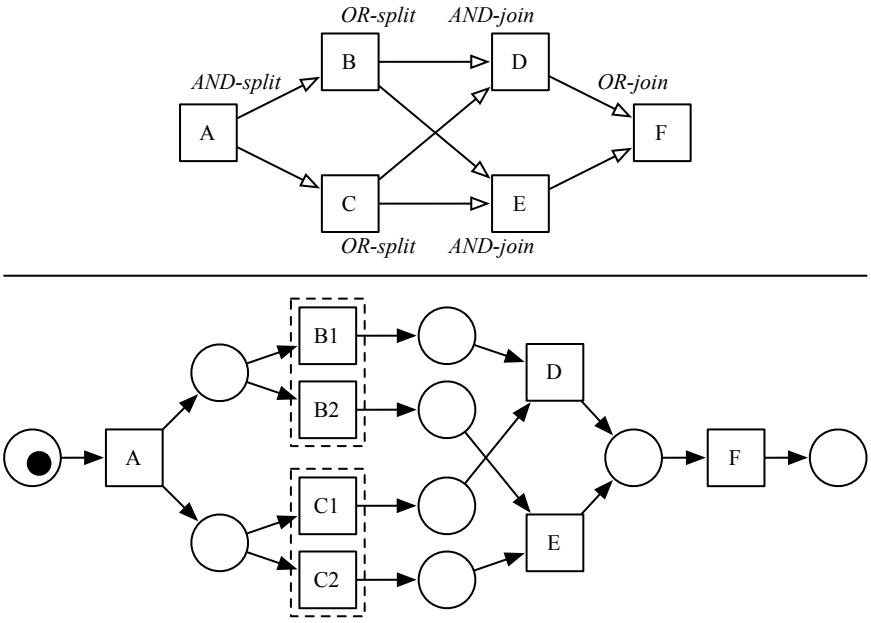


Figure 5.14: Example workflow (top) and corresponding Petri net (bottom) (van der Aalst, 1998b)

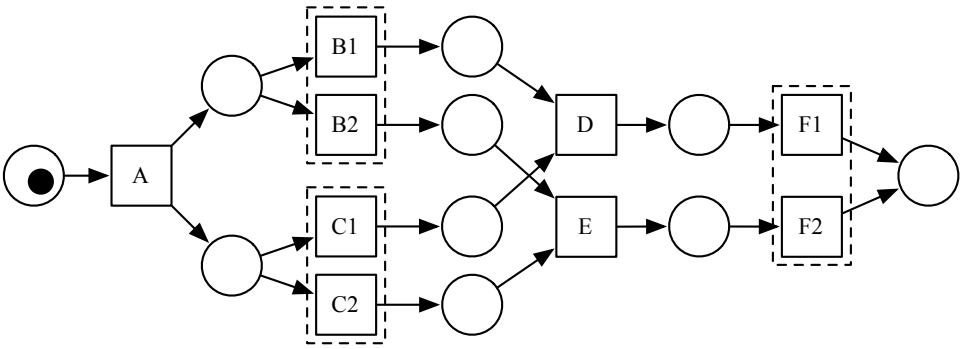


Figure 5.15: Expected corresponding Petri net for example workflow of Figure 5.14; note the OR-join pattern at the right of the figure

What we expect from our own Petri net semantics is therefore the following:

1. A precise algorithm for transforming a given workflow into its corresponding Petri net.
2. A straightforward way of determining the workflow element to which a given Petri net element corresponds.

In the following sections, we formally define UNIFY concerns as workflow graphs which have a corresponding Petri net, formally define UNIFY connectors as compositions of the connected concerns' Petri nets, and analyze the effects of these connectors on the correctness of workflows using *classical soundness* (van der Aalst, 2000) as the main correctness criterion. We owe our gratitude to Gilles Geeraerts, assistant professor at the Université Libre de Bruxelles and member of its formal methods and verification group, for introducing us to the domain of Petri nets and sharing some of his knowledge with us.

5.4.2 Petri Net Formalization of Concerns

5.4.2.1 Inductive Definition of UNIFY Workflow Graphs

As a starting point for our Petri net formalization of UNIFY workflows, we provide a number of definitions that formalize these workflows as directed graphs that consist of vertices that correspond to UNIFY Events, Activities, or ControlNodes, and edges that correspond to UNIFY Transitions. We define UNIFY workflow graphs inductively, as this will facilitate defining the corresponding Petri nets in Section 5.4.2.2. Because UNIFY's base language meta-model allows defining *arbitrary* workflows, i.e., workflows whose control flow is not restricted to a precisely defined set of control structures, we cannot base our inductive definition on these control structures, but rather introduce the intermediary notion of an *n–m fragment*: a fragment of a workflow graph which has n entry points and m exit points. We can then (re)define the single-entry single-exit (SESE) fragments we already introduced in Section 4.4.2 in terms of these n–m fragments, and define UNIFY workflow graphs in terms of these SESE fragments.

Definition 2 (n–m Fragment)**.** *An n–m fragment is a directed graph $\langle V, E, I, O\rangle$ where V is a finite set of vertices that are either events, activities, or control nodes, $E \subseteq V \times V$ is a set of edges, $I \subseteq V$ is a set of n input nodes that are all start events, and $O \subseteq V$ is a set of m output nodes that are all end events, such that one of the following holds:*

- *(Activity) $V = \{i, v, o\}$, $E = \{\langle i, v\rangle, \langle v, o\rangle\}$, $I = \{i\}$, and $O = \{o\}$, with i being a start event, v being an activity, and o being an end event*
- *(Sequence) There exists an n_1–m_1 fragment $\langle V_1, E_1, I_1, O_1\rangle$, an n_2–m_2 fragment $\langle V_2, E_2, I_2, O_2\rangle$, and a mapping $M_c \subseteq O_1 \times I_2 = \{\langle o_1^1, i_2^1\rangle, \ldots, \langle o_1^k, i_2^k\rangle\}$ that specifies where the fragments should be connected in sequence into an n–m fragment (with $n = n_1 + n_2 - k$ and $m = m_1 + m_2 - k$), such that:*
  - $V = (V_1 \cup V_2) \setminus \{o_1^i \in O_1 \mid i \in \{1, \ldots, k\}\} \setminus \{i_2^i \in I_2 \mid i \in \{1, \ldots, k\}\}$,

  - $E = \left(E_1 \cup E_2 \cup \{\langle v_1^i, v_2^i\rangle \in V_1 \times V_2 \mid \langle v_1^i, o_1^i\rangle \in E_1 \wedge \langle i_2^i, v_2^i\rangle \in E_2 \wedge i \in \{1,\ldots,k\}\}\right) \setminus \{\langle v_1^i, o_1^i\rangle \in E_1 \mid i \in \{1,\ldots,k\}\} \setminus \{\langle i_2^i, v_2^i\rangle \in E_2 \mid i \in \{1,\ldots,k\}\}$,
  - $I = (I_1 \cup I_2) \setminus \{i_2^i \in I_2 \mid i \in \{1,\ldots,k\}\}$, *and*
  - $O = (O_1 \cup O_2) \setminus \{o_1^i \in O_1 \mid i \in \{1,\ldots,k\}\}$

- *(Union)* $V = V_1 \cup V_2$, $E = E_1 \cup E_2$, $I = I_1 \cup I_2$, *and* $O = O_1 \cup O_2$, *with* $\langle V_1, E_1, I_1, O_1\rangle$ *and* $\langle V_2, E_2, I_2, O_2\rangle$ *being* n_1–m_1 *and* n_2–m_2 *fragments, respectively,* $n = n_1 + n_2$, *and* $m = m_1 + m_2$

- *(Split) There exists an* n_1–m_1 *fragment* $\langle V_1, E_1, I_1, O_1\rangle$, *a start event* i, *a split* v, *and a set* $\{i_1^1, \ldots, i_1^k\} \subset I_1$ *that specifies where the split should be connected to the fragment in order to obtain an n–m fragment (with* $n = n_1 + 1 - k$ *and* $m = m_1$*), such that:*

  - $V = (V_1 \cup \{i, v\}) \setminus \{i_1^i \in I_1 \mid i \in \{1,\ldots,k\}\}$,
  - $E = (E_1 \cup \{\langle i, v\rangle\} \cup \{\langle v, v_1^i\rangle \in \{v\} \times V_1 \mid \langle i_1^i, v_1^i\rangle \in E_1 \wedge i \in \{1,\ldots,k\}\}) \setminus \{\langle i_1^i, v_1^i\rangle \in E_1 \mid i \in \{1,\ldots,k\}\}$,
  - $I = (I_1 \cup \{i\}) \setminus \{i_1^i \in I_1 \mid i \in \{1,\ldots,k\}\}$, *and*
  - $O = O_1$

- *(Join) There exists an* n_1–m_1 *fragment* $\langle V_1, E_1, I_1, O_1\rangle$, *an end event* o, *a join* v, *and a set* $\{o_1^1, \ldots, o_1^k\} \subset O_1$ *that specifies where the join should be connected to the fragment in order to obtain an n–m fragment (with* $n = n_1$ *and* $m = m_1 + 1 - k$*), such that:*

  - $V = (V_1 \cup \{o, v\}) \setminus \{o_1^i \in O_1 \mid i \in \{1,\ldots,k\}\}$,
  - $E = (E_1 \cup \{\langle v, o\rangle\} \cup \{\langle v_1^i, v\rangle \in V_1 \times \{v\} \mid \langle v_1^i, o_1^i\rangle \in E_1 \wedge i \in \{1,\ldots,k\}\}) \setminus \{\langle v_1^i, o_1^i\rangle \in E_1 \mid i \in \{1,\ldots,k\}\}$,
  - $I = I_1$, *and*
  - $O = (O_1 \cup \{o\}) \setminus \{o_1^i \in O_1 \mid i \in \{1,\ldots,k\}\}$

Definition 3 (1–1 Fragment)**.** *A 1–1 fragment is an n–m fragment with 1 input node and 1 output node.*

Definition 4 (Workflow Graph)**.** *A workflow graph is a 1–1 fragment.*

5.4.2.2 Petri Net Formalization of UNIFY Workflow Graphs

A labeled Petri net is a tuple $N = \langle P, T, \Sigma\rangle$, where P is a finite set of *places*, T is a finite set of *transitions*, and Σ is a finite *alphabet*. Each transition $t \in T$ is a triple $\langle I_t, O_t, \ell\rangle$, where $I_t : P \mapsto \mathbb{N}$ and $O_t : P \mapsto \mathbb{N}$ are respectively *input* and *output* multisets of places, and $\ell \in \Sigma$ is the *transition label*.

Definition 5 (Petri Net Formalization of an n–m Fragment)**.** *Each n–m fragment as formalized in Section 5.4.2.1 has a corresponding labeled Petri net and a mapping* $M : V \mapsto (P \cup T)$ *of* UNIFY *workflow graph vertices to Petri net places or transitions, which are defined inductively as follows.*

- *(Activity) A fragment consisting of a start event i, a single activity v, and an end event o is represented by a labeled Petri net $N = \langle P, T, \Sigma \rangle$, with $P = \{p_i, p_o\}$, $T = \{t\}$ with $t = \langle \{p_i\}, \{p_o\}, v \rangle$, and $\Sigma = \{v\}$. The mapping of* UNIFY *workflow graph vertices to Petri net places or transitions is defined as $M = \{\langle i, p_i \rangle, \langle v, t \rangle, \langle o, p_o \rangle\}$.*

- *(Sequence) As is illustrated in Figure 5.16, a sequence of two fragments $\langle V_1, E_1, I_1, O_1 \rangle$ and $\langle V_2, E_2, I_2, O_2 \rangle$, which are represented by labeled Petri nets $N_1 = \langle P_1, T_1, \Sigma_1 \rangle$ and $N_2 = \langle P_2, T_2, \Sigma_2 \rangle$, respectively, which have mappings M_1 and M_2, respectively, and which are to be connected according to the mapping $M_c : O_1 \mapsto I_2 = \{\langle o_1^1, i_2^1 \rangle, \ldots, \langle o_1^k, i_2^k \rangle\}$, is represented by a labeled Petri net $N = \langle P, T, \Sigma \rangle$, with*

  - $P = P_1 \cup P_2 \setminus \{p \mid \langle i_2^i, p \rangle \in M_2 \wedge i \in \{1, \ldots, k\}\}$
  - $T = T_1 \cup T_2$, *where, for each $t = \langle I_t, O_t, \ell \rangle \in T_2$, for each $p \in I_t$, and for each $\langle i_2^i, p \rangle \in M_2$ with $i \in \{1, \ldots, k\}$, the set I_t is changed such that $p \notin I_t$ and $p' \in I_t$, with $\langle o_1^i, p' \rangle \in M_1 \wedge \langle o_1^i, i_2^i \rangle \in M_c$.*
  - $\Sigma = \Sigma_1 \cup \Sigma_2$

  The mapping of UNIFY *workflow graph vertices to Petri net places or transitions is defined as $M = M_1 \cup M_2 \setminus \{\langle o_1^i, p \rangle \mid \langle o_1^i, i_2^i \rangle \in M_c \wedge p \in P_1\} \setminus \{\langle i_2^i, p \rangle \mid \langle o_1^i, i_2^i \rangle \in M_c \wedge p \in P_2\}$.*

- *(Union) A union of two fragments $\langle V_1, E_1, I_1, O_1 \rangle$ and $\langle V_2, E_2, I_2, O_2 \rangle$, which are represented by labeled Petri nets $N_1 = \langle P_1, T_1, \Sigma_1 \rangle$ and $N_2 = \langle P_2, T_2, \Sigma_2 \rangle$, respectively, and which have mappings M_1 and M_2, respectively, is represented by a labeled Petri net $N = \langle P, T, \Sigma \rangle$, with $P = P_1 \uplus P_2$, $T = T_1 \uplus T_2$, and $\Sigma = \Sigma_1 \uplus \Sigma_2$. The mapping of* UNIFY *workflow graph vertices to Petri net places or transitions is defined as $M = M_1 \uplus M_2$.*

- *(AND-Split) A start event i and an AND-split s, which is to be connected to the start events $\{i_1^1, \ldots, i_1^k\}$ of a fragment $\langle V_1, E_1, I_1, O_1 \rangle$, which is represented by a labeled Petri net $N_1 = \langle P_1, T_1, \Sigma_1 \rangle$ and which has mapping M_1, are represented by a labeled Petri net $N = \langle P, T, \Sigma \rangle$, with*

  - $P = P_1 \cup \{p_s\}$
  - $T = T_1 \cup \{t_s\}$, *where $t_s = \langle \{p_s\}, \{p \mid \langle i_1^i, p \rangle \in M_1 \wedge i \in \{1, \ldots, k\}\}, s \rangle$*
  - $\Sigma = \Sigma_1 \cup \{s\}$

  The mapping of UNIFY *workflow graph vertices to Petri net places or transitions is defined as $M = M_1 \setminus \{\langle i_1^i, p \rangle \mid i \in \{1, \ldots, k\} \wedge p \in P_1\} \cup \{\langle i, p_s \rangle, \langle s, t_s \rangle\}$.*

- *(XOR-Split) A start event i and an XOR-split s, which is to be connected to the start events $\{i_1^1, \ldots, i_1^k\}$ of a fragment $\langle V_1, E_1, I_1, O_1 \rangle$, which is represented by a labeled Petri net $N_1 = \langle P_1, T_1, \Sigma_1 \rangle$ and which has mapping M_1, are represented by a labeled Petri net $N = \langle P, T, \Sigma \rangle$, with*

  - $P = P_1 \cup \{p_s\}$

  - $T = T_1 \cup \{t_s^i \mid t_s^i = \langle \{p_s\}, \{p\}, s_i \rangle \wedge \langle i_1^i, p \rangle \in M_1 \wedge i \in \{1, \ldots, k\}\}$
  - $\Sigma = \Sigma_1 \cup \{s_i \mid i \in \{1, \ldots, k\}\}$

  The mapping of UNIFY *workflow graph vertices to Petri net places or transitions is defined as* $M = M_1 \setminus \{\langle i_1^i, p \rangle \mid i \in \{1, \ldots, k\} \wedge p \in P_1\} \cup \{\langle i, p_s \rangle\} \cup \{\langle s, t_s^i \rangle \mid i \in \{1, \ldots, k\}\}$.

- *(AND-Join) An end event o and an AND-join j, which is to be connected to the end events* $\{o_1^1, \ldots, o_1^k\}$ *of a fragment* $\langle V_1, E_1, I_1, O_1 \rangle$*, which is represented by a labeled Petri net* $N_1 = \langle P_1, T_1, \Sigma_1 \rangle$ *and which has mapping* M_1*, are represented by a labeled Petri net* $N = \langle P, T, \Sigma \rangle$*, with*

  - $P = P_1 \cup \{p_j\}$
  - $T = T_1 \cup \{t_j\}$*, where* $t_j = \langle \{p \mid \langle o_1^i, p \rangle \in M_1 \wedge i \in \{1, \ldots, k\}\}, \{p_j\}, j \rangle$
  - $\Sigma = \Sigma_1 \cup \{j\}$

  The mapping of UNIFY *workflow graph vertices to Petri net places or transitions is defined as* $M = M_1 \setminus \{\langle o_1^i, p \rangle \mid i \in \{1, \ldots, k\} \wedge p \in P_1\} \cup \{\langle o, p_j \rangle, \langle j, t_j \rangle\}$.

- *(XOR-Join) An end event o and an XOR-join j, which is to be connected to the end events* $\{o_1^1, \ldots, o_1^k\}$ *of a fragment* $\langle V_1, E_1, I_1, O_1 \rangle$*, which is represented by a labeled Petri net* $N_1 = \langle P_1, T_1, \Sigma_1 \rangle$ *and which has mapping* M_1*, are represented by a labeled Petri net* $N = \langle P, T, \Sigma \rangle$*, with*

  - $P = P_1 \cup \{p_j\}$
  - $T = T_1 \cup \{t_j^i \mid t_j^i = \langle \{p\}, \{p_j\}, j_i \rangle \wedge \langle o_1^i, p \rangle \in M_1 \wedge i \in \{1, \ldots, k\}\}$
  - $\Sigma = \Sigma_1 \cup \{j_i \mid i \in \{1, \ldots, k\}\}$

  The mapping of UNIFY *workflow graph vertices to Petri net places or transitions is defined as* $M = M_1 \setminus \{\langle o_1^i, p \rangle \mid i \in \{1, \ldots, k\} \wedge p \in P_1\} \cup \{\langle o, p_j \rangle\} \cup \{\langle j, t_j^i \rangle \mid i \in \{1, \ldots, k\}\}$.

As UNIFY workflow graphs are defined in terms of 1–1 fragments, and 1–1 fragments are defined in terms of n–m fragments, the above definition constitutes a full formalization of UNIFY workflow graphs using Petri nets. In conclusion of this formalization, let us briefly discuss the above definition:

- Due to the *Activity* rule, every activity is represented by a Petri net transition labeled with the activity's name. The transition has one input place and one output place which represent the start and end events of the fragment.
- The *Sequence* rule does not add any new places or transitions, but rather removes the input places which represent the start events of the second fragment where the first fragment should be connected, and replaces these input places by the output places which represent the end events of the first fragment. Thus, the graph edges connecting the two fragments are represented in the Petri net as the former output places of the first fragment.
- The *Union* rule adds nor removes places or transitions.

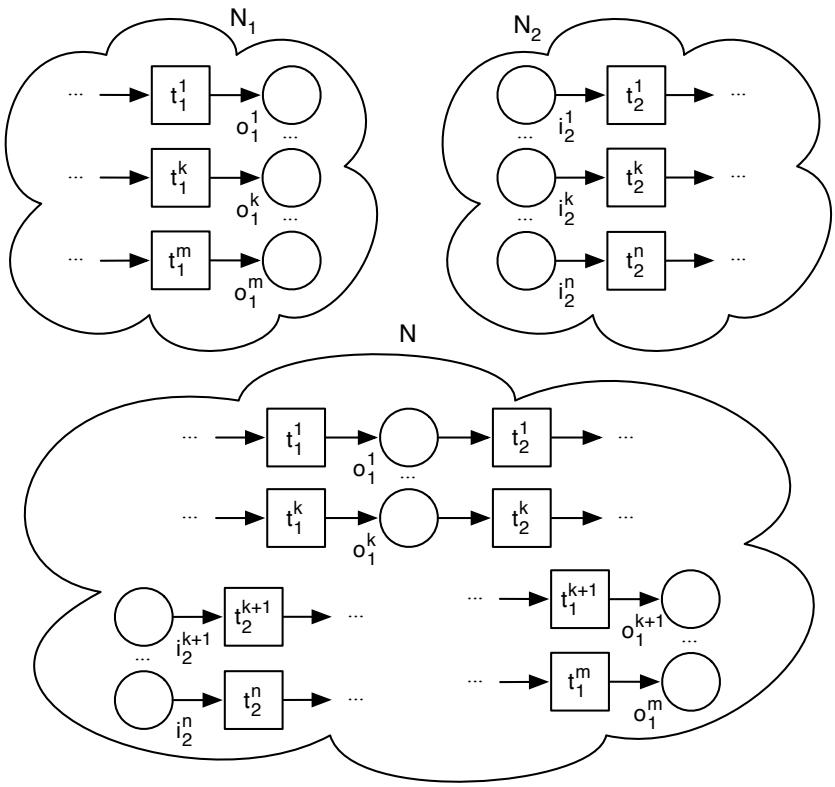


Figure 5.16: Construction of a Petri net N as a sequence of two Petri nets N_1 and N_2 according to the mapping $M_c : O_1 \mapsto I_2 = \{\langle o_1^1, i_2^1\rangle, \ldots, \langle o_1^k, i_2^k\rangle\}$

- Due to the *AND-split* and *AND-join* rules, every AND-split and AND-join is represented by a single Petri net transition labeled with the node's name.
  - An AND-split's transition has a new input place representing the new fragment's start event, while its output places are the input places of the old fragment to which the split should be connected. These places thus represent the graph edges connecting the split to the nodes of the old fragment.
  - An AND-join's transition has a new output place representing the new fragment's end event, while its input places are the output places of the old fragment to which the join should be connected. These places thus represent the graph edges connecting the join to the nodes of the old fragment.
- Due to the *XOR-split* and *XOR-join* rules, every XOR-split and XOR-join is represented by k Petri net transitions labeled with the node's name (k being the number of outgoing/incoming edges of the XOR-split/XOR-join).

- Each of a XOR-split's transitions has as its input place a new place representing the new fragment's start event, while each transition's single output place is each of the input places of the old fragment to which the split should be connected. These places thus represent the graph edges connecting the split to the nodes of the old fragment.
- Each of a XOR-join's transitions has as its output place a new place representing the new fragment's end event, while each transition's single input place is each of the output places of the old fragment to which the join should be connected. These places thus represent the graph edges connecting the join to the nodes of the old fragment.

Informally, our translation from UNIFY base language primitives to Petri net elements can be visualized as in Figure 5.17. Note that this translation is compatible with the Petri net patterns for workflow primitives of van der Aalst (1998b), which were already shown in Figure 5.13. Any UNIFY workflow can thus be translated into a Petri net using the following algorithm:

1. For each UNIFY transition in the workflow, a Petri net place is generated. While doing this, a mapping from UNIFY transitions to their corresponding Petri net places is constructed, which will be used in the following steps of the algorithm.

2. For each UNIFY activity in the workflow, a Petri net transition is generated. The Petri net transition's input place is the place that corresponds to the UNIFY activity's incoming transition. The Petri net transition's output place is the place that corresponds to the UNIFY activity's outgoing transition.

3. For each UNIFY AND-split in the workflow, a Petri net transition is generated. The Petri net transition's input place is the place that corresponds to the UNIFY AND-split's incoming transition. The Petri net transition's output places are the places that correspond to the UNIFY AND-split's outgoing transitions.

4. For each UNIFY AND-join in the workflow, a Petri net transition is generated. The Petri net transition's input places are the places that correspond to the UNIFY AND-join's incoming transitions. The Petri net transition's output place is the place that corresponds to the UNIFY AND-join's outgoing transition.

5. For each UNIFY XOR-split's outgoing transition, a Petri net transition is generated. The Petri net transition's input place is the place that corresponds to the UNIFY XOR-split's incoming transition. The Petri net transition's output place is the place that corresponds to the UNIFY XOR-split's outgoing transition.

6. For each UNIFY XOR-join's incoming transition, a Petri net transition is generated. The Petri net transition's input place is the place that corresponds to the UNIFY XOR-join's incoming transition. The Petri net transition's output place is the place that corresponds to the UNIFY XOR-split's outgoing transition.

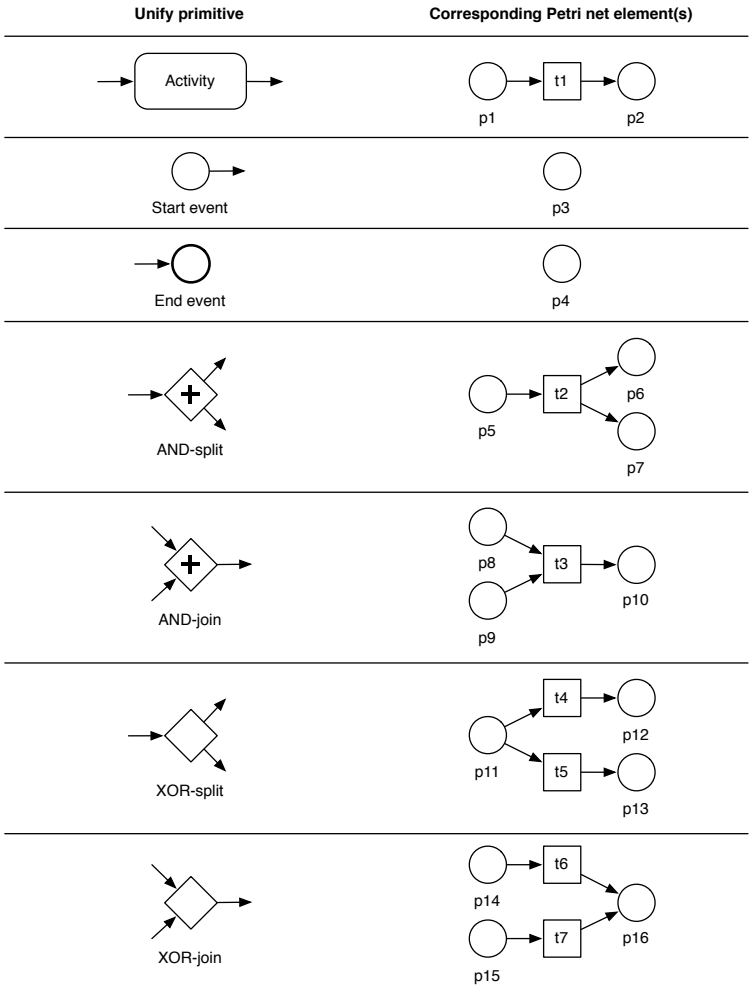


Figure 5.17: Mapping from UNIFY base language primitives to Petri net elements

Using this algorithm, it is straightforward to transform a given workflow into its corresponding Petri net. While performing steps 2–6 of the algorithm, it is possible to augment the mapping from UNIFY transitions to Petri net places constructed in step 1 with a mapping from UNIFY activities, AND-splits, AND-joins, XOR-splits, and XOR-joins to their corresponding Petri net transitions. Using this augmented mapping, any Petri net element can easily be traced back to the UNIFY base language primitive to which it corresponds. We have implemented the transformation algorithm as part of the UNIFY framework, and allow exporting the resulting Petri net models as PNML files, PNML being a standard for the exchange of Petri nets supported by existing Petri net editors and analysis tools.

Figure 5.18 illustrates the execution of the algorithm for an example UNIFY workflow, which is shown at the top of Figure 5.18 and is equivalent to the workflow at the top of Figure 5.14. Note that the generated Petri net contains more transitions and places than the Petri net at the bottom of Figure 5.14, because each AND-split, AND-join, XOR-split, and XOR-join gives rise to "dummy" transitions, i.e., transitions that do not represent a workflow activity. However, now we have a clear algorithm for the transformation of a workflow into a Petri net, and we have a means of tracing Petri net elements back to workflow elements.

5.4.3 Petri Net Formalization of Connectors

We can now introduce a formalization of UNIFY connectors based purely on Petri nets. We expect this formalization to provide the following benefits:

1. We aim to allow analyzing the effects of UNIFY connectors on a workflow's operational properties without requiring the entire workflow composition to be translated to Petri nets. Thus, we will obtain a more compositional approach that is better suited to traditional Petri net analysis tools, which may have trouble dealing with large Petri nets.

2. Although our current experiments with UNIFY have primarily used source code weaving for the implementation of our connector mechanism, and thus ensure compatibility with existing tool chains, runtime weaving is a popular strategy in general aspect-oriented research. Given that Petri nets offer a natural runtime semantics for workflows, a Petri net formalization of UNIFY connectors forms a basis for implementing runtime weaving of connectors.

We aim to ensure that our Petri net formalization of UNIFY connectors is equivalent with the Graph Transformation formalization we presented in Section 5.3. An intuition for this statement can be gained by comparing the Petri net composition rules we present in the remainder of this section with the graph transformation rules we presented in Section 5.3.2. A formal proof for the equivalence of both formalizations is beyond our current scope.

For the definition of the semantics of UNIFY connectors, we start from:

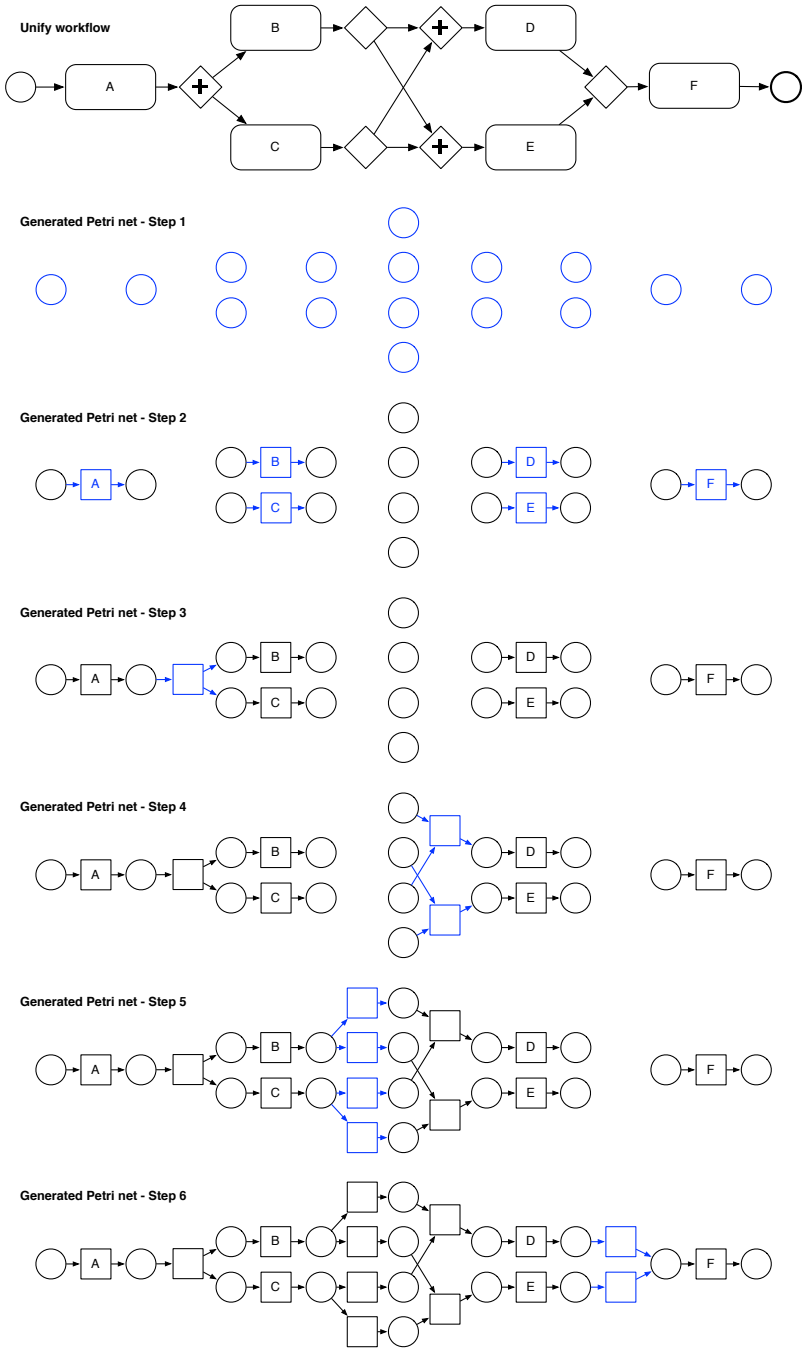


Figure 5.18: Construction of the Petri net that corresponds to the example workflow at the top of Figure 5.14

- A finite set of Petri nets $P = \{N_{W1}, \ldots, N_{Wn}\}$ of which each Petri net $N_{Wi} = \langle P_i, T_i, \Sigma_i \rangle$ represents a UNIFY workflow W_i as defined in Definition 5. This is the set of Petri nets to which connectors can apply.

- A finite set of connectors $\{C_1, \ldots, C_m\}$ of which each connector C_i is a triple $\langle N_{Wa}, type, places \rangle$, with $N_{Wa} = \langle P_a, T_a, \Sigma_a \rangle$ being a Petri net representing the advice, $type \in \{before, after, replace, around, parallel, alternative, iterating, synchronizing, switching\}$ being the type of the connector, and $places = \langle p_s, p_e \rangle$ being a couple of places which are the start and end places of a (1–1) joinpoint fragment.

When applying a connector, the pointcut expression of a connector C_i will resolve to a set of fragments, which each can be identified by a start and end place, that determine where the advice N_{Wa} will be inserted. The location where the advice is inserted relative to the joinpoint is determined by the connector type. In the remainder of this section, we describe for each type of connector how the advice is inserted relative to a single joinpoint fragment.

5.4.3.1 Sequential Concern Connection Patterns

Before Consider the connector $C_{before} = \langle N_{Wa}, before, \langle p_s, p_e \rangle \rangle$ and the net $N_{Wx} = \langle P_x, T_x, \Sigma_x \rangle \in P$ containing a joinpoint fragment identified by the start and end places $\langle p_s, p_e \rangle$. The Petri net $N_{Wa} = \langle P_a, T_a, \Sigma_a \rangle$ has as start place p'_s and as end place p'_e.

The composed Petri net $N_{Wc} = \langle P_c, T_c, \Sigma_c \rangle$ is constructed as follows (cf. Figure 5.19):

- $P_c = P_a \cup P_x$
- $T_c = T_a \cup T'_x \cup \{t_b\}$, where:
  - $t'_a = \langle I_a, O'_a, a \rangle \in T'_x \iff t_a = \langle I_a, O_a, a \rangle \in T_x$, where

$$O'_a = \begin{cases} O_a \setminus \{p_s\} \cup \{p'_s\} & \text{if } p_s \in O_a \\ O_a & \text{if } p_s \notin O_a \end{cases}$$

  - $t_b = \langle I_b, O_b, b \rangle$, where $I_b = \{p'_e\}$ and $O_b = \{p_s\}$
- $\Sigma_c = \Sigma_a \cup \Sigma_x \cup \{b\}$

After Consider the connector $C_{after} = \langle N_{Wa}, after, \langle p_s, p_e \rangle \rangle$ and the net $N_{Wx} = \langle P_x, T_x, \Sigma_x \rangle \in P$ containing a joinpoint fragment identified by the start and end places $\langle p_s, p_e \rangle$. The Petri net $N_{Wa} = \langle P_a, T_a, \Sigma_a \rangle$ has as start place p'_s and as end place p'_e.

The composed Petri net $N_{Wc} = \langle P_c, T_c, \Sigma_c \rangle$ is constructed as follows (cf. Figure 5.20):

- $P_c = P_a \cup P_x$
- $T_c = T_a \cup T'_x \cup \{t_a\}$, where:

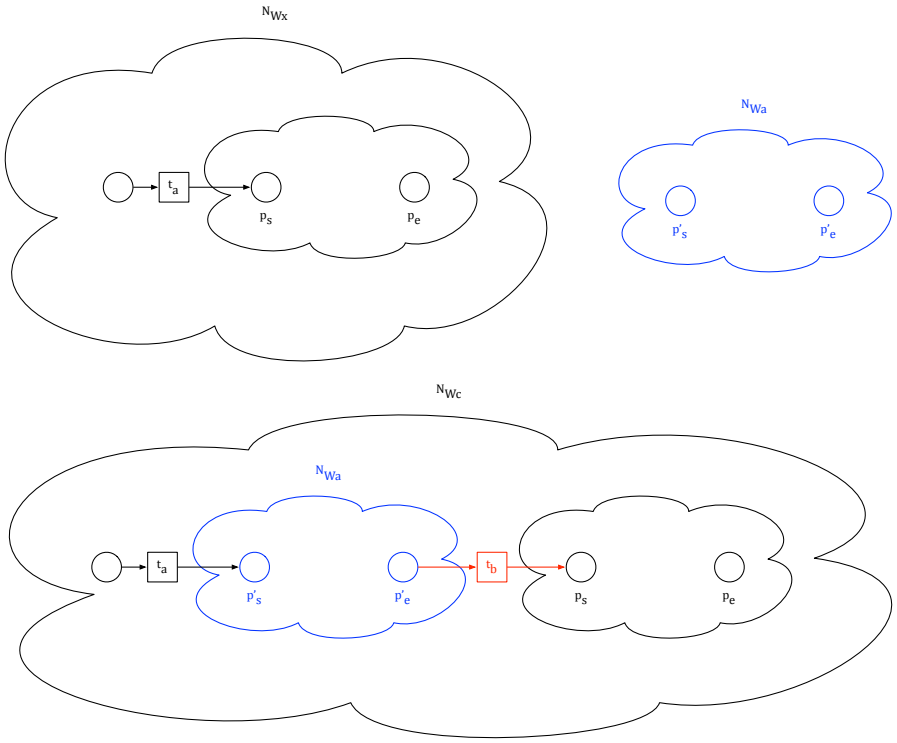


Figure 5.19: Construction of N_{Wc} by applying C_{before} to N_{Wx}

- $t'_b = \langle I'_b, O_b, b\rangle \in T'_x \iff t_b = \langle I_b, O_b, b\rangle \in T_x$, where

$$I'_b = \begin{cases} I_b \setminus \{p_e\} \cup \{p'_e\} & \text{if } p_e \in I_b \\ I_b & \text{if } p_e \notin I_b \end{cases}$$

- $t_a = \langle I_a, O_a, a\rangle$, where $I_a = \{p_e\}$ and $O_a = \{p'_s\}$

- $\Sigma_c = \Sigma_a \cup \Sigma_x \cup \{a\}$

Replace Consider the connector $C_{replace} = \langle N_{Wa}, replace, \langle p_s, p_e\rangle\rangle$ and the net $N_{Wx} = \langle P_x, T_x, \Sigma_x\rangle \in P$ containing a joinpoint fragment characterized by the Petri net $N_j = \langle P_j, T_j, \Sigma_j\rangle$ and identified by the start and end places $\langle p_s, p_e\rangle$. The Petri net $N_{Wa} = \langle P_a, T_a, \Sigma_a\rangle$ has as start place p'_s and as end place p'_e.

The composed Petri net $N_{Wc} = \langle P_c, T_c, \Sigma_c\rangle$ is constructed as follows (cf. Figure 5.21):

- $P_c = (P_a \cup P_x) \setminus P_j$

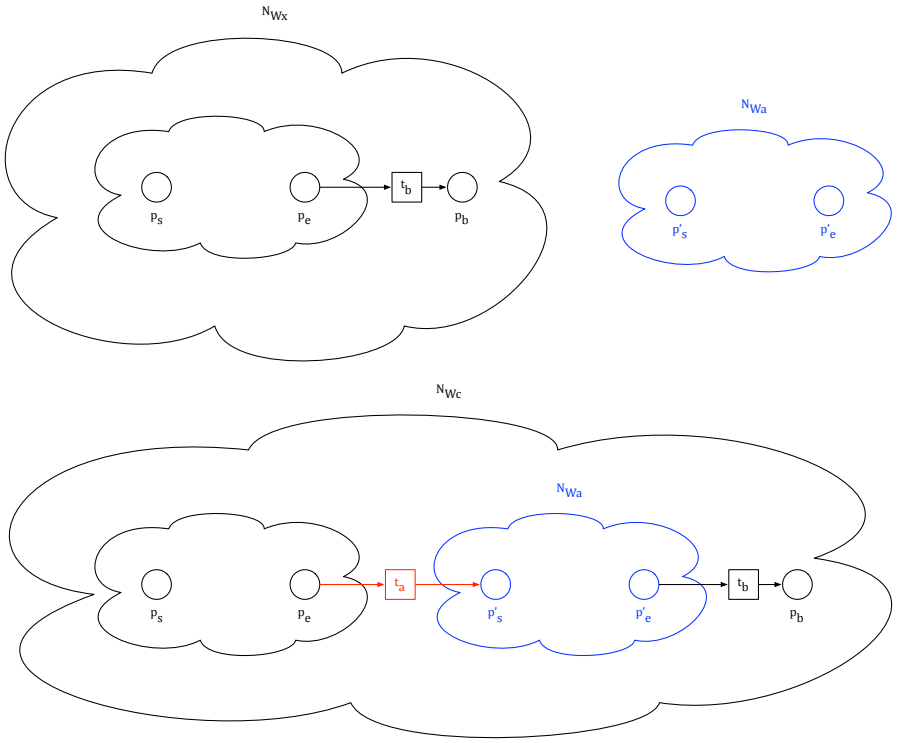


Figure 5.20: Construction of N_{Wc} by applying C_{after} to N_{Wx}

- $T_c = (T_a \cup T_x) \setminus T_j$ where for each $t_a = \langle I_a, O_a, a \rangle \in T_x$ with $p_s \in O_a$ the set O_a is changed such that p_s is replaced by p'_s, and for each $t_b = \langle I_b, O_b, b \rangle \in T_x$ with $p_e \in I_b$ the set I_b is changed such that p_e is replaced by p'_e
- $\Sigma_c = (\Sigma_a \cup \Sigma_x) \setminus \Sigma_j$

Around Consider the connector $C_{around} = \langle N_{Wa}, around, \langle p_s, p_e \rangle \rangle$ and the net $N_{Wx} = \langle P_x, T_x, \Sigma_x \rangle \in P$ containing a joinpoint fragment characterized by the Petri net $N_j = \langle P_j, T_j, \Sigma_j \rangle$ and identified by the start and end places $\langle p_s, p_e \rangle$. The Petri net $N_{Wa} = \langle P_a, T_a, \Sigma_a \rangle$ has as start place p'_s and as end place p'_e and contains a proceed fragment characterized by the Petri net $N_p = \langle P_p, T_p, \Sigma_p \rangle$ and identified by the start and end places $\langle p''_s, p''_e \rangle$.

The composed Petri net $N_{Wc} = \langle P_c, T_c, \Sigma_c \rangle$ is constructed as follows (cf. Figure 5.22):

- $P_c = (P_a \cup P_x) \setminus P_p$
- $T_c = (T_a \cup T_x) \setminus T_p$, where:

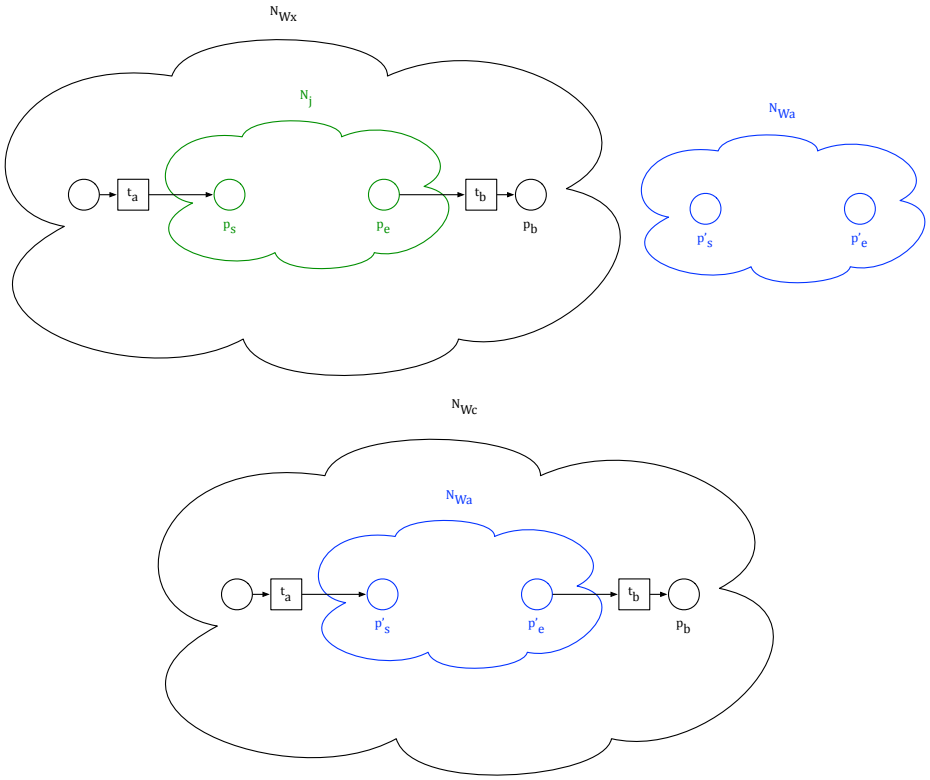


Figure 5.21: Construction of N_{Wc} by applying $C_{replace}$ to N_{Wx}

- for each $t_a = \langle I_a, O_a, a \rangle \in T_x$ with $p_s \in O_a$ the set O_a is changed such that p_s is replaced by p'_s
- for each $t_b = \langle I_b, O_b, b \rangle \in T_x$ with $p_e \in I_b$ the set I_b is changed such that p_e is replaced by p'_e
- for each $t_c = \langle I_c, O_c, c \rangle \in T_a$ with $p''_s \in O_c$ the set O_c is changed such that p''_s is replaced by p_s
- for each $t_d = \langle I_d, O_d, d \rangle \in T_a$ with $p''_e \in I_d$ the set I_d is changed such that p''_e is replaced by p_e

- $\Sigma_c = (\Sigma_a \cup \Sigma_x) \setminus \Sigma_p$

5.4.3.2 Parallel Concern Connection Patterns

Parallel Consider the connector $C_{parallel} = \langle N_{Wa}, parallel, \langle p_s, p_e \rangle \rangle$ and the net $N_{Wx} = \langle P_x, T_x, \Sigma_x \rangle \in P$ containing a joinpoint fragment identified by the start and end places $\langle p_s, p_e \rangle$. The Petri net $N_{Wa} = \langle P_a, T_a, \Sigma_a \rangle$ has as start place p'_s and as end place p'_e.

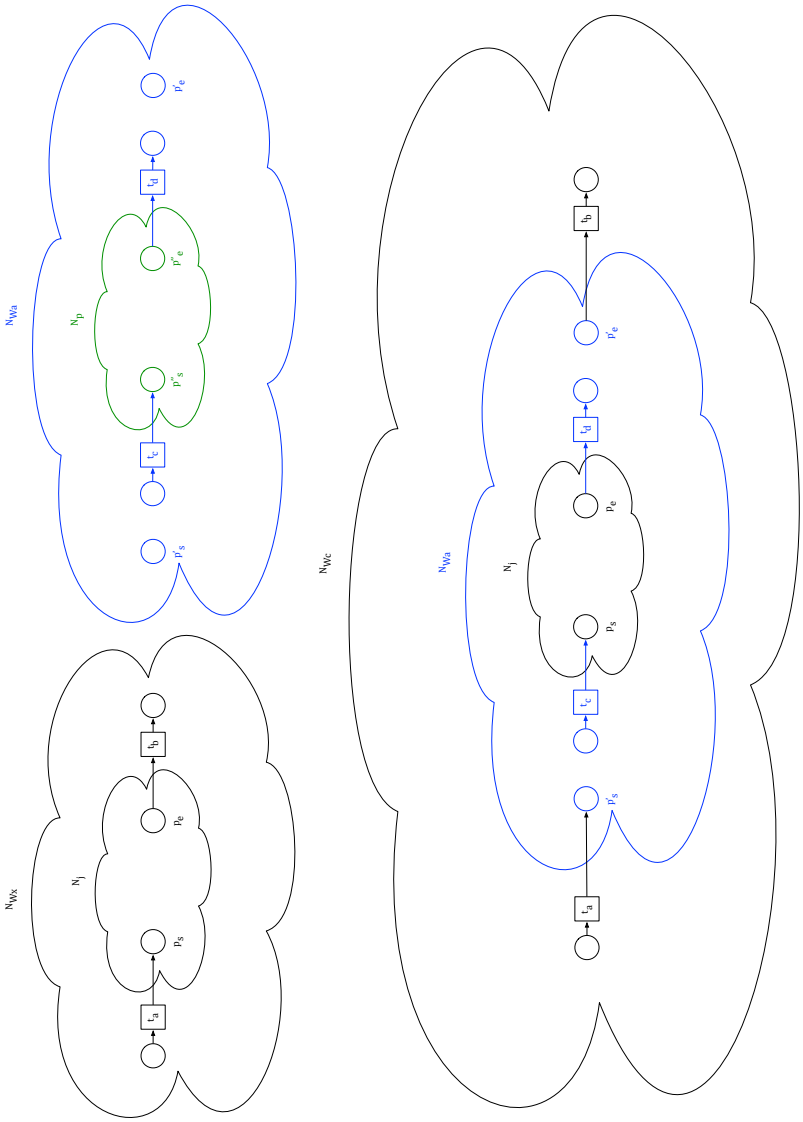


Figure 5.22: Construction of N_{Wc} by applying C_{around} to N_{Wx}

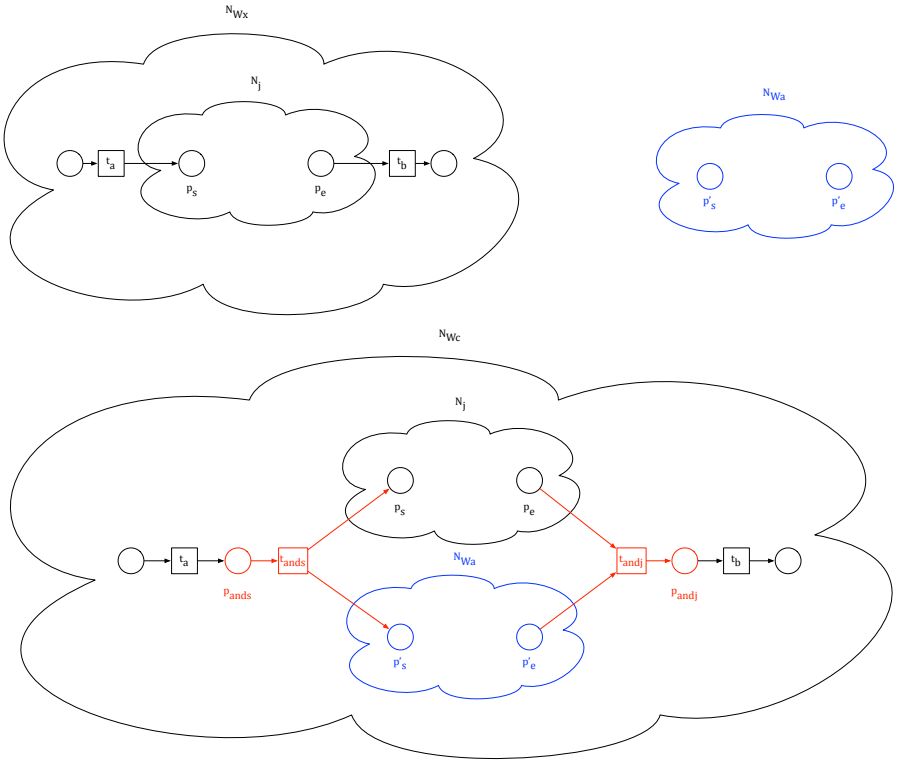


Figure 5.23: Construction of N_{Wc} by applying $C_{parallel}$ to N_{Wx}

The composed Petri net $N_{Wc} = \langle P_c, T_c, \Sigma_c \rangle$ is constructed as follows (cf. Figure 5.23):

- $P_c = P_a \cup P_x \cup \{p_{ands}, p_{andj}\}$
- $T_c = T_a \cup T_x \cup \{t_{ands}, t_{andj}\}$, where:
  - for each $t_a = \langle I_a, O_a, a \rangle \in T_x$ with $p_s \in O_a$, the set O_a is changed such that p_s is replaced by p_{ands}
  - for each $t_b = \langle I_b, O_b, b \rangle \in T_x$ with $p_e \in I_a$, the set I_a is changed such that p_e is replaced by p_{andj}
  - $t_{ands} = \langle I_{ands}, O_{ands}, ands \rangle$, where $I_{ands} = \{p_{ands}\}$ and $O_{ands} = \{p_s, p'_s\}$
  - $t_{andj} = \langle I_{andj}, O_{andj}, andj \rangle$, where $I_{andj} = \{p_e, p'_e\}$ and $O_{andj} = \{p_{andj}\}$
- $\Sigma_c = \Sigma_a \cup \Sigma_x \cup \{ands, andj\}$

5.4.3.3 Conditional Concern Connection Patterns

Alternative Consider the connector $C_{alternative} = \langle N_{Wa}, alternative, \langle p_s, p_e \rangle \rangle$ and the net $N_{Wx} = \langle P_x, T_x, \Sigma_x \rangle \in P$ containing a joinpoint fragment identified by the start and end places $\langle p_s, p_e \rangle$. The Petri net $N_{Wa} = \langle P_a, T_a, \Sigma_a \rangle$ has as start place p'_s and as end place p'_e.

The composed Petri net $N_{Wc} = \langle P_c, T_c, \Sigma_c \rangle$ is constructed as follows (cf. Figure 5.24):

- $P_c = P_a \cup P_x \cup \{p_{xors}, p_{xorj}\}$
- $T_c = T_a \cup T_x \cup \{t_{xors1}, t_{xors2}, t_{xorj1}, t_{xorj2}\}$ where
  - for each $t_a = \langle I_a, O_a, a \rangle \in T_x$ with $p_s \in O_a$, the set O_a is changed such that p_s is replaced by p_{xors}
  - for each $t_b = \langle I_b, O_b, b \rangle \in T_x$ with $p_e \in I_b$, the set I_b is changed such that p_e is replaced by p_{xorj}
  - $t_{xors1} = \langle I_{xors1}, O_{xors1}, xors1 \rangle$, where $I_{xors1} = \{p_{xors}\}$ and $O_{xors1} = \{p_s\}$
  - $t_{xors2} = \langle I_{xors2}, O_{xors2}, xors2 \rangle$, where $I_{xors2} = \{p_{xors}\}$ and $O_{xors2} = \{p'_s\}$
  - $t_{xorj1} = \langle I_{xorj1}, O_{xorj1}, xorj1 \rangle$, where $I_{xorj1} = \{p_e\}$ and $O_{xorj1} = \{p_{xorj}\}$
  - $t_{xorj2} = \langle I_{xorj2}, O_{xorj2}, xorj2 \rangle$, where $I_{xorj2} = \{p'_e\}$ and $O_{xorj2} = \{p_{xorj}\}$
- $\Sigma_c = \Sigma_a \cup \Sigma_x \cup \{xors1, xors2, xorj1, xorj2\}$

5.4.3.4 Iterating Concern Connection Patterns

Iterating Consider the connector $C_{iterating} = \langle N_{Wa}, iterating, \langle p_s, p_e \rangle \rangle$ and the net $N_{Wx} = \langle P_x, T_x, \Sigma_x \rangle \in P$ containing a joinpoint fragment identified by the start and end places $\langle p_s, p_e \rangle$. The Petri net $N_{Wa} = \langle P_a, T_a, \Sigma_a \rangle$ has as start place p'_s and as end place p'_e.

The composed Petri net $N_{Wc} = \langle P_c, T_c, \Sigma_c \rangle$ is constructed as follows (cf. Figure 5.25):

- $P_c = P_a \cup P_x \cup \{p_{xorj}, p_{xors}\}$,
- $T_c = T_a \cup T_x \cup \{t_{js}, t_{je}, t_{se}, t_{ss}\}$, where:
  - $t_{js} = \langle I_{js}, O_{js}, js \rangle$, where $I_{js} = \{p_{xorj}\}$ and $O_{js} = \{p_s\}$
  - $t_{je} = \langle I_{je}, O_{je}, je \rangle$, where $I_{je} = \{p'_e\}$ and $O_{je} = \{p_s\}$
  - $t_{se} = \langle I_{se}, O_{se}, se \rangle$, where $I_{se} = \{p_e\}$ and $O_{se} = \{p_{xors}\}$,
  - $t_{ss} = \langle I_{ss}, O_{ss}, ss \rangle$, where $I_{ss} = \{p_e\}$ and $O_{ss} = \{p'_s\}$
  - for each $t_a = \langle I_a, O_a, a \rangle \in T_x$ with $p_s \in O_a$ the set O_a is changed such that p_s is replaced by p_{xorj}
  - for each $t_b = \langle I_b, O_b, b \rangle \in T_x$ with $p_e \in I_b$ the set I_b is changed such that p_e is replaced by p_{xors}
- $\Sigma_c = \Sigma_a \cup \Sigma_x \cup \{js, je, se, ss\}$

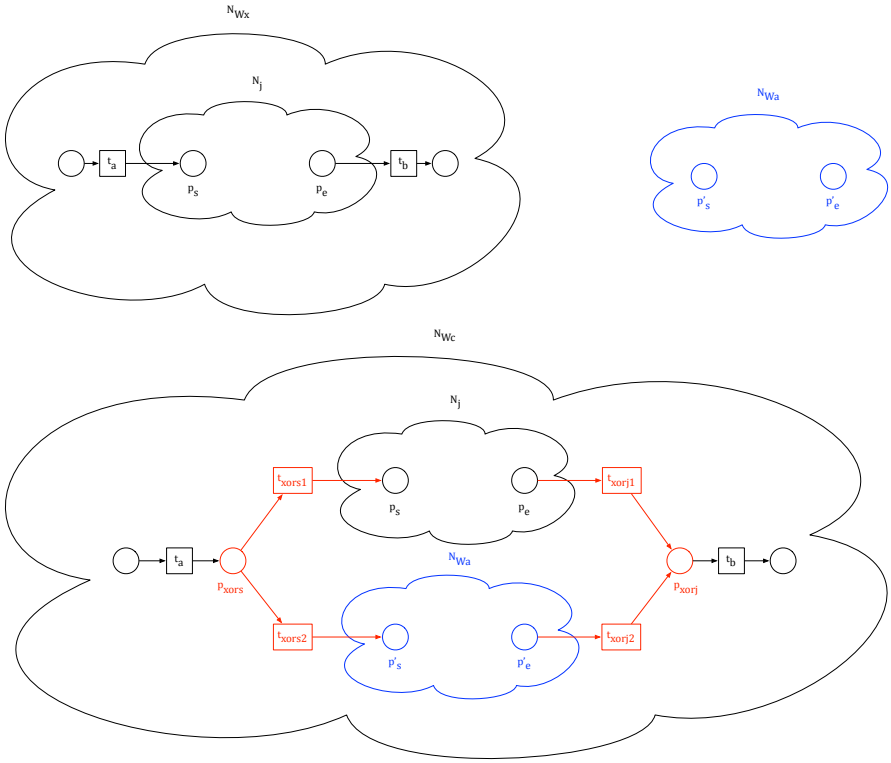


Figure 5.24: Construction of N_{Wc} by applying $C_{alternative}$ to N_{Wx}

5.4.3.5 Internal Concern Connection Patterns

Synchronizing Consider the connector $C_{synchronizing} = \langle N_{Wa}, synchronizing, \langle p_s, p_e \rangle \rangle$ and the net $N_{Wx} = \langle P_x, T_x, \Sigma_x \rangle \in P$ containing a joinpoint fragment identified by the start and end places $\langle p_s, p_e \rangle$. This fragment contains two other fragments, one identified by the start place p''_s and the end place p''_e, and another identified by the start place p'''_s and the end place p'''_e. The Petri net $N_{Wa} = \langle P_a, T_a, \Sigma_a \rangle$ has as start place p'_s and as end place p'_e.

The composed Petri net $N_{Wc} = \langle P_c, T_c, \Sigma_c \rangle$ is constructed as follows (cf. Figure 5.26):

- $P_c = P_a \cup P_x \cup \{p_{ands}, p_{andj}\}$
- $T_c = T_a \cup T_x \cup \{t_{ands}, t_{andj}\}$, where:
  - $t_{ands} = \langle I_{ands}, O_{ands}, ands \rangle$, where $I_{ands} = \{p''_e\}$ and $O_{ands} = \{p_{ands}, p'_s\}$
  - for each $t_a = \langle I_a, O_a, a \rangle \in T_x$ with $p''_e \in I_a$ the set I_a is changed such that p''_e is replaced by p_{ands}
  - $t_{andj} = \langle I_{andj}, O_{andj}, andj \rangle$, where $I_{andj} = \{p'_e, p_{andj}\}$ and $O_{andj} = p'''_s$

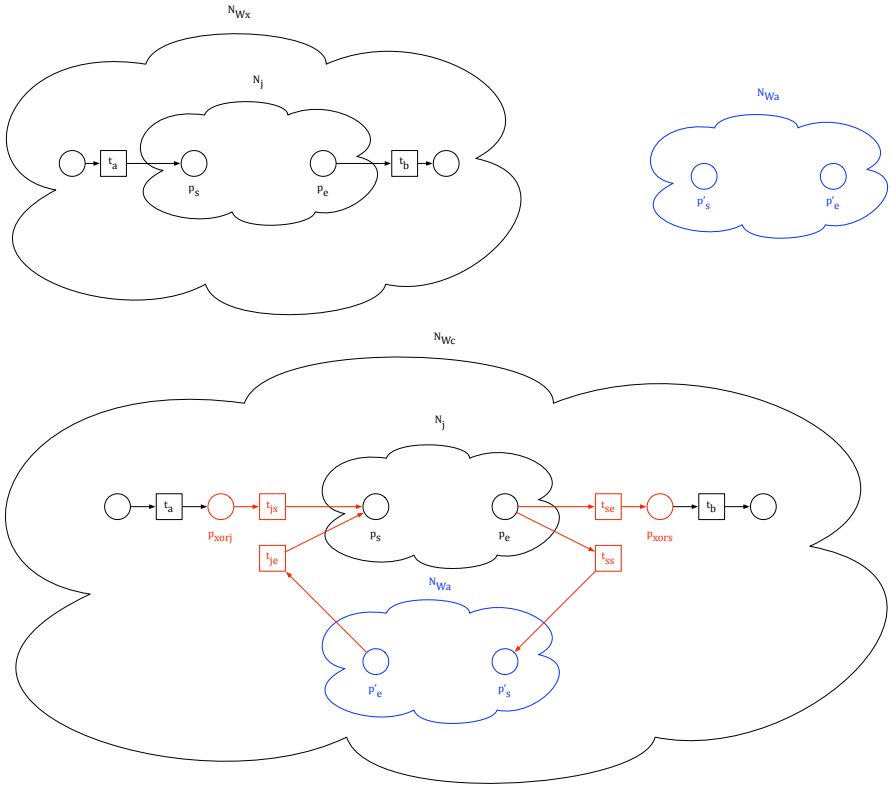


Figure 5.25: Construction of N_{Wc} by applying $C_{iterating}$ to N_{Wx}

  - for each $t_b = \langle I_b, O_b, b \rangle \in T_x$ with $p_s''' \in O_b$ the set O_b is changed such that p_s''' is replaced by p_{andj}

- $\Sigma_c = \Sigma_a \cup \Sigma_x \cup \{ands, andj\}$

Switching Consider the connector $C_{switching} = \langle N_{Wa}, switching, \langle p_s, p_e \rangle \rangle$ and the net $N_{Wx} = \langle P_x, T_x, \Sigma_x \rangle \in P$ containing a joinpoint fragment identified by the start and end places $\langle p_s, p_e \rangle$. This fragment contains two other fragments, one identified by the start place p_s'' and the end place p_e'', and another identified by the start place p_s''' and the end place p_e'''. The Petri net $N_{Wa} = \langle P_a, T_a, \Sigma_a \rangle$ has as start place p_s' and as end place p_e'.

The composed Petri net $N_{Wc} = \langle P_c, T_c, \Sigma_c \rangle$ is constructed as follows (cf. Figure 5.27):

- $P_c = P_a \cup P_x \cup \{p_{xors}, p_{xorj}\}$

- $T_c = T_a \cup T_x \cup \{t_{xors1}, t_{xors2}, t_{xorj1}, t_{xorj2}\}$, where:

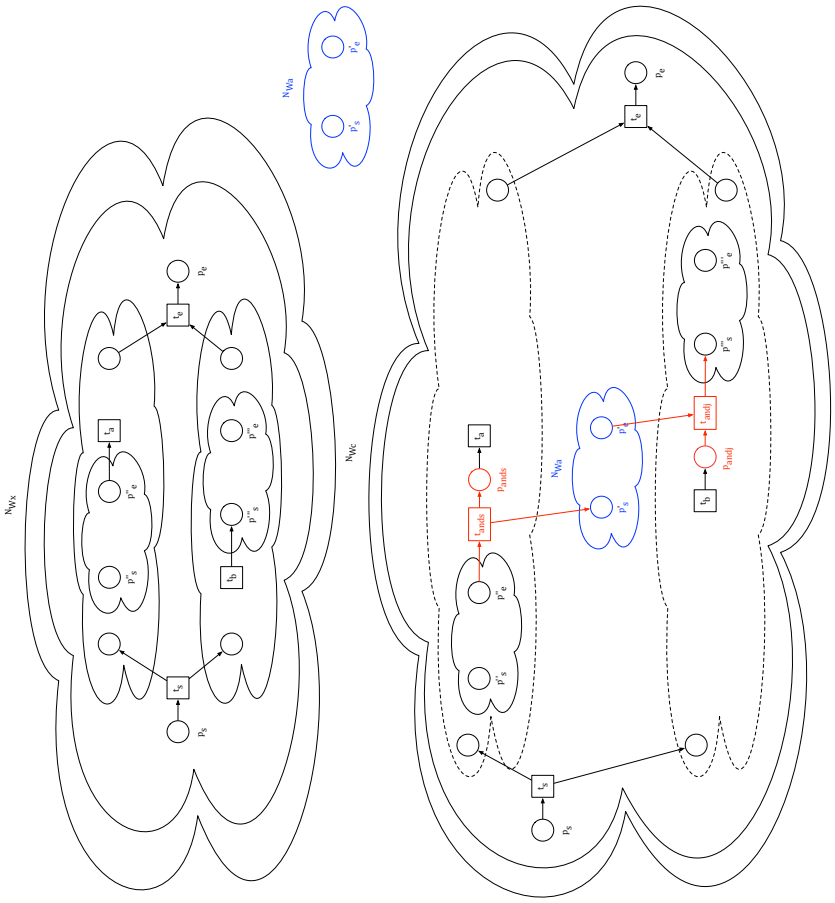


Figure 5.26: Construction of N_{Wc} by applying $C_{synchronizing}$ to N_{Wx}

  - $t_{xors1} = \langle I_{xors1}, O_{xors1}, xors1 \rangle$, where $I_{xors1} = \{p''_e\}$ and $O_{xors1} = \{p_{xors}\}$
  - $t_{xors2} = \langle I_{xors2}, O_{xors2}, xors2 \rangle$, where $I_{xors2} = \{p''_e\}$ and $O_{xors2} = \{p'_s\}$
  - for each $t_a = \langle I_a, O_a, a \rangle \in T_x$ with $p''_e \in I_a$ the set I_a is changed such that p''_e is replaced by p_{xors}
  - $t_{xorj1} = \langle I_{xorj1}, O_{xorj1}, xorj1 \rangle$, where $I_{xorj1} = \{p'_e\}$ and $O_{xorj1} = \{p'''_s\}$
  - $t_{xorj2} = \langle I_{xorj2}, O_{xorj2}, xorj2 \rangle$, where $I_{xorj2} = \{p_{xorj}\}$ and $O_{xorj2} = \{p'''_s\}$
  - for each $t_b = \langle I_b, O_b, b \rangle \in T_x$ with $p'''_s \in O_b$ the set O_b is changed such that p'''_s is replaced by p_{xorj}

- $\Sigma_c = \Sigma_a \cup \Sigma_x \cup \{xors1, xors2, xorj1, xorj2\}$

5.4.4 Analysis

Now that we have a clear Petri net specification of both UNIFY concerns and connectors, we can start analyzing the composition of UNIFY concerns into a workflow using the various connectors. Our main goal here is to prove that our composition mechanism does not introduce any undesirable properties into the composed workflow, i.e., that the composed workflows are correct. Of course, this requires a clearly specified correctness criterion. Within the workflow community, there is a consensus around using the *soundness* property as the main criterion for correctness of workflows (van der Aalst et al., 2011). This property guarantees the absence of deadlocks (a case gets stuck), livelocks (a case cannot progress), and other anomalies that can be detected without domain knowledge. Before we provide a formal definition of soundness, let us briefly enumerate some notations used throughout this section:

- $[p]$ denotes the state (or *marking*) in which only the place p is marked with a token.
- $M_i \xrightarrow{t} M_j$ denotes that state M_j is reachable from state M_i by firing a single transition t.
- $M_1 \xrightarrow{*} M_n$ denotes that state M_n is reachable from state M_1 by sequentially firing an arbitrary number of transitions.
- $M_1 \xrightarrow{\sigma} M_n$ denotes that state M_n is reachable from state M_1 by sequentially firing a specific sequence of transitions named σ.
- $M_1 \xrightarrow[N]{\sigma} M_n$ denotes that state M_n is reachable from state M_1 in a specific Petri net N by sequentially firing a specific sequence of transitions named σ.
- $M\big|_{P_x}$ denotes the projection of state M on the set of places P_x.

Although there exist a number of alternative definitions of soundness (which are surveyed in van der Aalst et al., 2011), we will use the definition of *classical soundness* by van der Aalst (2000):

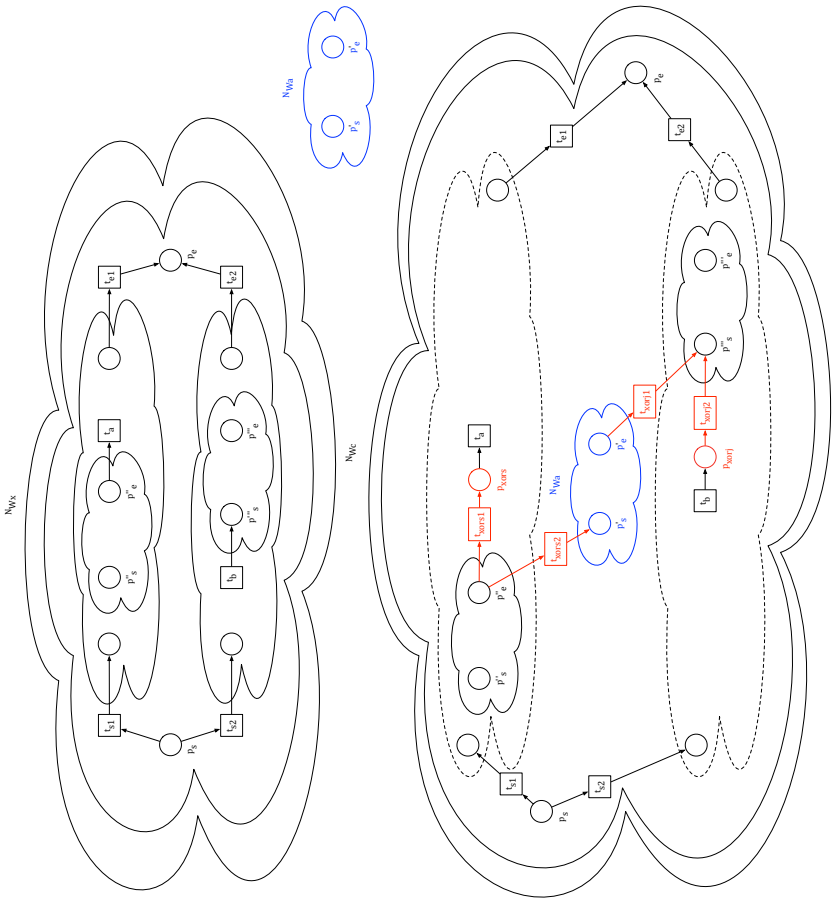


Figure 5.27: Construction of N_{Wc} by applying $C_{switching}$ to N_{Wx}

Definition 6 (Classical Soundness)**.** *A workflow modeled by a Petri net $N = \langle P, T, \Sigma \rangle$ is sound if and only if:*

1. *(Option to Complete) For every state M reachable from the initial state $[i]$, there exists a firing sequence leading from state M to the final state $[o]$. Formally:*

$$\forall M : ([i] \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} [o])$$

2. *(Proper Completion) The final state $[o]$ is the only state reachable from the initial state $[i]$ with at least one token in the final place o. Formally:*

$$\forall M : ([i] \xrightarrow{*} M \wedge M \geq [o]) \Rightarrow (M = [o]))$$

3. *(No Dead Transitions) There are no dead transitions in N with initial state $[i]$. Formally:*

$$\forall t \in T \; \exists M, M' : [i] \xrightarrow{*} M \xrightarrow{t} M'$$

We can thus refine the goal of this section to be the following: **we aim to enable proving that a connector, which connects a classically sound advice to a classically sound workflow, gives rise to a composed workflow that is classically sound.** Before presenting the outline of a prototypical proof, we will consider three important properties of the Petri net compositions defined in Section 5.4.3, which can be established by induction on the possible firing sequences of these Petri nets. In these properties, we assume a situation similar to Figure 5.19, in which p_s and p_e are the initial and final places of the Petri net representing the joinpoint, respectively, p'_s and p'_e are the initial and final places of the Petri net representing the advice, respectively, and t_a and t_b are the transitions of the Petri net representing the composed workflow that lead the Petri net representing the base workflow into and out of the Petri net representing the advice, respectively (i.e., the firing of t_a indicates the "activation" of the advice in the composed workflow, whereas the firing of t_b indicates its "deactivation").

Property A. *Let M be a marking such that $[i] \xrightarrow[N_{Wc}]{\sigma} M$, i.e., M is reachable in N_{Wc} from its initial marking $[i]$ by some firing sequence σ. Assume $\sigma = t_1 \ldots t_n$. Let j (resp. k) denote the largest position in σ such that $t_j = t_a$ ($t_k = t_b$). Assume $t_j = 0$ ($t_k = 0$) if t_a (t_b) does not occur in σ. (Intuitively, j is the "last time" the advice has been entered in σ, and k is the last time the advice has been left in σ.)*

1. *If $k \geq j$ (i.e., the advice is "inactive" in M), then $M\big|_{P_x}$ is reachable in N_{Wx}.*
2. *If $k < j$ (i.e., the advice is active in M), then $M\big|_{P_a}$ is reachable in N_{Wa}, and M' is reachable in N_{Wx} where $\forall p \in P_x \setminus \{p_s\} : M'(p) = M(p) \wedge M'(p_s) = 1$ (i.e., M' is the marking we would have reached if we would not have entered the advice).*

Property B. ***If*** *σ is fireable from M in N_{Wa}* ***then*** *σ is fireable from all M' in N_{Wc} such that $\forall p \in P_a : M'(p) = M(p)$. Moreover,*

$$M \xrightarrow[N_{Wa}]{\sigma} M_1 \text{ implies } M' \xrightarrow[N_{Wc}]{\sigma} M'_1 \text{ such that } \forall p : M'_1(p) = \begin{cases} M_1(p) & \text{if } p \in P_a \\ M'(p) & \text{if } p \notin P_a \end{cases}$$

Property C. ***If*** *σ is fireable from M in N_{Wx}* ***and*** *σ does not contain t_a* ***then*** *σ is fireable from all M' in N_{Wc} such that $\forall p \in P_x : M'(p) = M(p)$. Moreover,*

$$M \xrightarrow[N_{Wx}]{\sigma} M_1 \textit{ implies } M' \xrightarrow[N_{Wc}]{\sigma} M'_1 \textit{ such that } \forall p : M'_1(p) = \begin{cases} M_1(p) & \textit{if } p \in P_x \\ M'(p) & \textit{if } p \notin P_x \end{cases}$$

In general, one can say that these properties formalize the relation between the markings and firing sequences of the base workflow's Petri net, the advice's Petri net, and the composed Petri net. The properties are useful when proving the soundness of a composed Petri net when the soundness of the base workflow's Petri net and the advice's Petri net is assumed. We will now outline a proof for each of UNIFY's connectors. We start by introducing a prototypical proof for the soundness of a connector, and exemplify it using the before connector. The main idea is to prove each of the three necessary and sufficient conditions for classical soundness as defined in Definition 6 (option to complete, proper completion, and no dead transitions) by considering possible markings and firing sequences of the composed Petri net.

5.4.4.1 Soundness Proof for the Before Connector

Option to Complete Let M be a marking in N_{Wc} such that $[i] \xrightarrow{\sigma} M$ (with $\sigma = t_1 \dots t_n$). Then we must prove that there exists a σ' in N_{Wc} such that $M \xrightarrow{\sigma'} [o]$. Let j (resp. k) be the largest position in σ such that $t_j = t_a$ ($t_k = t_b$)

1. If $k \geq j$ (i.e., the advice is inactive in M)
   - Since either the advice has never been activated in σ ($j = k = 0$) or it has been activated but has finished ($k > j > 0$), and since the advice respects **proper completion**, we know that $\forall p \in P_a : M(p) = 0$
   - By Property A, $M\big|_{P_x}$ is reachable in N_{Wx}
   - Since N_{Wx} respects **option to complete**, there exists a $\overline{\sigma}$ in N_{Wx} such that $M\big|_{P_x} \xrightarrow{\overline{\sigma}} [o]$
   - Let θ be the sequence of transitions obtained from $\overline{\sigma}$ by replacing each occurrence of t_a by $t_a \sigma_a t_b$, where σ_a is a sequence of transitions of N_{Wa} such that $[p'_s] \xrightarrow{\sigma_a} [p'_e]$ (such a σ_a always exists because N_{Wa} respects **option to complete** and **proper completion**).
     That is if $\overline{\sigma} = \sigma_1 t_a \sigma_2 t_a \dots t_a \sigma_k$ where all σ_i do not contain t_a, then $\theta = \sigma_1 t_a \sigma_a t_b \sigma_2 t_a \sigma_a t_b \dots t_a \sigma_a t_b \sigma_k$
   - It is easy to show that θ is fireable in N_{Wc} (due to Properties A, B, C) and that it reaches $[o]$
2. If $k < j$ (i.e., the advice is active in M)
   - By Property A, $M\big|_{P_a}$ is reachable in N_{Wa}

- Since N_{Wa} respects **option to complete**, there exists a sequence σ' in N_{Wa} such that $M\big|_{P_a} \xrightarrow{\sigma'} [p'_e]$
- By Property B, σ' is fireable in N_{Wc}, and

$$M \xrightarrow{\sigma'} M' \text{ implies } \begin{cases} M'(p'_e) = 1 & \\ M'(p) = 0 & \forall\, p \in P_a \setminus \{p'_e\} \\ M'(p) = M(p) & \forall\, p \in P_x \end{cases}$$

- Thus, t_b is fireable from M' and moves the token from p'_e to p_s
- From there, we continue as in (1)

Proper Completion Let M be a marking that is reachable in N_{Wc} such that $M \geq [o]$. Then we must prove that $M = [o]$. Assume $[i] \xrightarrow{\sigma} M$, with $\sigma = \sigma_1 t_a \theta_1 t_b \sigma_2 t_a \theta_2 t_b \ldots \sigma_n t_a \theta_n t_b \sigma_{n+1}$, every θ_i giving rise to marking M_i, and every subsequent t_b giving rise to marking M'_i

- $\forall i : M_i \geq p'_e$. Since N_{Wa} respects **proper completion**, $M_i\big|_{P_a} = p'_e$, thus $\forall\, p \in P_a$: $M'_i(p) = 0$, $\forall\, p \in P_x \setminus \{p_s\} : M'_i(p) = M_i(p)$, and $M'_i(p_s) = 1$
- By Property A, $M'_n\big|_{P_x}$ is reachable in N_{Wx} and σ_{n+1} is fireable in N_{Wx} (because $\forall\, p \in P_a : M'_n(p) = 0$), and $M'_n\big|_{P_x} \xrightarrow{\sigma_{n+1}} M\big|_{P_x}$
- By hypothesis, $M \geq [o]$, and thus $M\big|_{P_x} \geq [o]$ because $o \in P_x$
- By hypothesis, N_{Wx} respects **proper completion** such that $M\big|_{P_x} = [o]$
- Moreover, since $\forall\, p \in P_a : M'_n(p) = 0$ and σ_{n+1} does not contain t_a, $\forall\, p \in P_a$: $M(p) = 0$
- Therefore, $M = [o]$

No Dead Transitions Given a transition $t \in T_c$, we must find a sequence σ such that $[i] \xrightarrow{\sigma} M \xrightarrow{t} M'$

1. If $t \in T_x$
   - Since N_{Wx} respects **no dead transitions**, we know that there exists a θ such that $[i] \xrightarrow[N_{Wx}]{\theta} M \xrightarrow{t} M'$
   - From θ, we can build $\overline{\theta}$ by replacing each occurrence of t_a by $t_a \overline{\sigma} t_b$ where $\overline{\sigma}$ is a sequence of the advice such that $[p'_s] \xrightarrow[N_{Wa}]{\overline{\sigma}} [p'_e]$, which exists because N_{Wa} respects **option to complete**.
   - By Properties B and C, $\overline{\theta}$ is fireable in N_{Wc} and reaches $\overline{M}$ such that $\overline{M}\big|_{P_x} = M$

2. If $t \in T_a$

   - We know that t_a is not dead in N_{Wx} because it respects **no dead transitions**.
   - By the same reasoning as in case 1, we can build $\overline{\theta}$ such that $[i] \xrightarrow[N_{Wc}]{\overline{\theta}} M'' \xrightarrow{t_a} M'''$ with $M'''(p'_s) = 1$
   - Since t is not dead in N_{Wa} because it respects **no dead transitions**, there exists a τ such that $[p'_s] \xrightarrow[N_{Wa}]{\tau} M \xrightarrow{t} M'$
   - By Properties B and C, $\overline{\theta}\, t_a \tau$ is fireable in N_{Wc} and reaches M such that $M \xrightarrow{t} M'$

Q.E.D.

5.4.4.2 Soundness Proof for the After Connector

The proof for the after connector is completely analogous to the proof for the before connector presented in Section 5.4.4.1; the only difference being that instead of referring to the joinpoint's incoming transition t_a within the base workflow's Petri net N_{Wx}, the proof refers to the joinpoint's outgoing transition t_b. We therefore do not provide the complete proof at this point in the dissertation, but rather refer the interested reader to Appendix B.

5.4.4.3 Soundness Proof for the Replace Connector

The proof for the replace connector is roughly analogous to the proof for the before connector. Nevertheless, there are important differences due to the need to remove the joinpoint N_j from the base workflow N_{Wx} (cf. Figure 5.21 on page 131). In a given firing sequence of N_{Wx}, we can detect the entering and exiting of the joinpoint by investigating the occurrences of transitions t_a and t_b, similar to how we detected the entering and exiting of the advice in the properties and proofs presented above. However, between the firing of t_a and t_b, there may be an interleaving of transitions of T_j and transitions of $T_x \setminus T_j$. For example, consider the following firing sequence leading from the joinpoint's initial state to its final state, with all $t_{ji} \in T_j$ and all $t_{xi} \in T_x \setminus T_j$:

$$[p_s,\ldots] \xrightarrow{t_{j1}} M_1 \xrightarrow{t_{x1}} M_2 \xrightarrow{t_{x2}} M_3 \xrightarrow{t_{j2}} M_4 \xrightarrow{t_{x3}} M_5 \xrightarrow{t_{j3}} M_6 \xrightarrow{t_{j4}} [p_e,\ldots]$$

It is essential to the current proof that we recognize that, because the joinpoint of a replace connector is a single-entry single-exit fragment, t_a and t_b are the only transitions of N_{Wx} leading into and out of the joinpoint, and the transitions of T_j thus commute with the transitions of $T_x \setminus T_j$. This means that, with respect to the reachable markings before entering and after exiting the joinpoint, the example firing sequence is equivalent to:

$$[p_s,\ldots] \xrightarrow{t_{j1}} M'_1 \xrightarrow{t_{j2}} M'_2 \xrightarrow{t_{j3}} M'_3 \xrightarrow{t_{j4}} M'_4 \xrightarrow{t_{x1}} M'_5 \xrightarrow{t_{x2}} M'_6 \xrightarrow{t_{x3}} [p_e,\ldots]$$

We can now proceed with the actual proof for the replace connector.

Option to Complete Let M be a marking in N_{Wc} such that $[i] \xrightarrow{\sigma} M$ (with $\sigma = t_1 \dots t_n$). Then we must prove that there exists a σ' in N_{Wc} such that $M \xrightarrow{\sigma'} [o]$. Let j (resp. k) be the largest position in σ such that $t_j = t_a$ ($t_k = t_b$)

1. If $k \geq j$ (i.e., the advice is inactive in M)

   - Since either the advice has never been activated in σ ($j = k = 0$) or it has been activated but has finished ($k > j > 0$), and since the advice respects **proper completion**, we know that $\forall p \in P_a : M(p) = 0$
   - By Property A, $M\big|_{P_x}$ is reachable in N_{Wx}
   - Since N_{Wx} respects **option to complete**, there exists a $\overline{\sigma}$ in N_{Wx} such that $M\big|_{P_x} \xrightarrow{\overline{\sigma}} [o]$
   - Consider the sequence of transitions $\overline{\sigma}$, in which each pair of transitions t_a and t_b surrounds an interleaving ϕ_i of transitions of T_j and $T_x \setminus T_j$. We can reorder each ϕ_i in a way that places all transitions of T_j at the front of the sequence, resulting in a $\overline{\sigma}'$ that contains a number of occurrences of $t_a \phi_{ji} \phi_{xi} t_b$, where ϕ_{ji} is the sequence of transitions of ϕ_i belonging to T_j, and ϕ_{xi} is the sequence of transitions of ϕ_i belonging to $T_x \setminus T_j$.
   - Let θ be the sequence of transitions obtained from $\overline{\sigma}'$ by replacing each occurrence of $t_a \phi_{ji} \phi_{xi} t_b$ by $t_a \sigma_a \phi_{xi} t_b$, where σ_a is a sequence of transitions of N_{Wa} such that $[p'_s] \xrightarrow{\sigma_a} [p'_e]$ (such a σ_a always exists because N_{Wa} respects **option to complete** and **proper completion**).[5]
     That is if $\overline{\sigma}' = \sigma_1 t_a \phi_{j1} \phi_{x1} t_b \sigma_2 t_a \phi_{j2} \phi_{x2} t_b \dots t_a \phi_{jk-1} \phi_{xk-1} t_b \sigma_k$ where all σ_i do not contain t_a (or t_b), then $\theta = \sigma_1 t_a \sigma_a \phi_{x1} t_b \sigma_2 t_a \sigma_a \phi_{x2} t_b \dots t_a \sigma_a \phi_{xk-1} t_b \sigma_k$
   - It is easy to show that θ is fireable in N_{Wc} (due to Properties A, B, C) and that it reaches $[o]$

2. If $k < j$ (i.e., the advice is active in M)

   - By Property A, $M\big|_{P_a}$ is reachable in N_{Wa}
   - Since N_{Wa} respects **option to complete**, there exists a sequence σ' in N_{Wa} such that $M\big|_{P_a} \xrightarrow{\sigma'} [p'_e]$
   - By Property B, σ' is fireable in N_{Wc}, and

$$M \xrightarrow{\sigma'} M' \text{ implies } \begin{cases} M'(p'_e) = 1 & \\ M'(p) = 0 & \forall p \in P_a \setminus \{p'_e\} \\ M'(p) = M(p) & \forall p \in P_x \end{cases}$$

   - Thus, t_b is fireable from M' and moves the token from p'_e to p_b
   - From there, we continue as in (1)

[5] Essentially, we extract all the firing sequences of T_j out of $\overline{\sigma}'$, and insert a valid firing sequence of the advice after each occurrence of t_a.

Proper Completion Let M be a marking that is reachable in N_{Wc} such that $M \geq [o]$. Then we must prove that $M = [o]$. Assume $[i] \xrightarrow{\sigma} M$, with $\sigma = \sigma_1 t_a \theta_1 t_b \sigma_2 t_a \theta_2 t_b \ldots \sigma_n t_a \theta_n t_b \sigma_{n+1}$, every θ_i giving rise to marking M_i, and every subsequent t_b giving rise to marking M'_i

- $\forall i : M_i \geq p'_e$. Since N_{Wa} respects **proper completion**, $M_i\big|_{P_a} = p'_e$, thus $\forall p \in P_a : M'_i(p) = 0$, $\forall p \in P_x \setminus \{p_b\} : M'_i(p) = M_i(p)$, and $M'_i(p_b) = 1$
- By Property A, $M'_n\big|_{P_x}$ is reachable in N_{Wx} and σ_{n+1} is fireable in N_{Wx} (because $\forall p \in P_a : M'_n(p) = 0$), and $M'_n\big|_{P_x} \xrightarrow{\sigma_{n+1}} M\big|_{P_x}$
- By hypothesis, $M \geq [o]$, and thus $M\big|_{P_x} \geq [o]$ because $o \in P_x$
- By hypothesis, N_{Wx} respects **proper completion** such that $M\big|_{P_x} = [o]$
- Moreover, since $\forall p \in P_a : M'_n(p) = 0$ and σ_{n+1} does not contain t_a, $\forall p \in P_a : M(p) = 0$
- Therefore, $M = [o]$

No Dead Transitions Given a transition $t \in T_c$, we must find a sequence σ such that $[i] \xrightarrow{\sigma} M \xrightarrow{t} M'$

1. If $t \in T_x$
   - Since N_{Wx} respects **no dead transitions**, we know that there exists a θ such that $[i] \xrightarrow[N_{Wx}]{\theta} M \xrightarrow{t} M'$
   - Consider the sequence of transitions θ, in which each pair of transitions t_a and t_b surrounds an interleaving ϕ_i of transitions of T_j and $T_x \setminus T_j$. We can reorder each ϕ_i in a way that places all transitions of T_j at the front of the sequence, resulting in a θ' that contains a number of occurrences of $t_a \phi_{ji} \phi_{xi} t_b$, where ϕ_{ji} is the sequence of transitions of ϕ_i belonging to T_j, and ϕ_{xi} is the sequence of transitions of ϕ_i belonging to $T_x \setminus T_j$.
   - From θ', we can build $\overline{\theta}$ by replacing each occurrence of $t_a \phi_{ji} \phi_{xi} t_b$ by $t_a \overline{\sigma} \phi_{xi} t_b$ where $\overline{\sigma}$ is a sequence of the advice such that $[p'_s] \xrightarrow[N_{Wa}]{\overline{\sigma}} [p'_e]$, which exists because N_{Wa} respects **option to complete**.[6]
   - By Properties B and C, $\overline{\theta}$ is fireable in N_{Wc} and reaches $\overline{M}$ such that $\overline{M}\big|_{P_x} = M$
2. If $t \in T_a$
   - We know that t_a is not dead in N_{Wx} because it respects **no dead transitions**.

[6] Essentially, we extract all the firing sequences of T_j out of θ', and insert a valid firing sequence of the advice after each occurrence of t_a.

- By the same reasoning as in case 1, we can build $\overline{\theta}$ such that $[i] \xrightarrow[N_{Wc}]{\overline{\theta}} M'' \xrightarrow{t_a} M'''$ with $M'''(p'_s) = 1$
- Since t is not dead in N_{Wa} because it respects **no dead transitions**, there exists a τ such that $[p'_s] \xrightarrow[N_{Wa}]{\tau} M \xrightarrow{t} M'$
- By Properties B and C, $\overline{\theta}\, t_a \tau$ is fireable in N_{Wc} and reaches M such that $M \xrightarrow{t} M'$

Q.E.D. It should now be clear that the proof follows the same general strategy as the proofs for the before and after connectors, with the notable exception of reordering interleavings in the joinpoint while proving the option to complete and no dead transitions.

5.4.4.4 Soundness Proof for the Around Connector

Consider the around connector's Petri net semantics as illustrated by Figure 5.22 (on page 132). Similar to the proof for the replace connector, we can adapt our prototypical proof to deal with the possible interleavings of transitions of T_j, transitions of $T_x \setminus T_j$, and transitions of $T_a \setminus T_p$. However, we can simplify our proof by considering that the application of an around connector amounts to two applications of a replace connector. Given the base workflow N_{Wx} containing the joinpoint fragment N_j and the advice workflow N_{Wa} containing the proceed fragment N_p, all of which are assumed to be sound, the composed workflow N_{Wc} can be constructed as follows:

1. Consider N_{Wa} as illustrated at the top right of Figure 5.22. Apply a replace connector to N_{Wa} (which thus acts as a base workflow) in order to replace fragment N_p (which acts as the joinpoint) by fragment N_j (which acts as the advice). Assuming the soundness of N_{Wa}, N_j, and N_p, the proof presented in Section 5.4.4.3 guarantees the soundness of the composed Petri net N'_{Wa}.
2. Apply a replace connector to N_{Wx} in order to replace fragment N_j by N'_{Wa}. Assuming the soundness of N_{Wx} and N_j, and having proven the soundness of N'_{Wa}, the proof presented in Section 5.4.4.3 guarantees the soundness of the composed Petri net $N'_{Wx} = N_{Wc}$.

5.4.4.5 Soundness Proof for the Parallel Connector

Consider the parallel connector's Petri net semantics as illustrated by Figure 5.23 (on page 133). Once again, we can adapt our prototypical proof to deal with the possible interleavings of transitions of different parts of the composed Petri net between each firing of t_a and each subsequent firing of t_b. Whereas the replace connector essentially extracted all the firing sequences of T_j and inserted a valid firing sequence of the advice after each occurrence of t_a, the parallel connector does not extract any firing sequences but only inserts t_{ands}, a valid firing sequence of the advice, and t_{andj} at the appropriate locations between t_a and t_b. Once again, such a proof benefits from the fact that the

joinpoint and the advice are single-entry single-exit fragments, and the transitions of T_j, the transitions of $T_x \setminus T_j$, and the transitions of T_a thus commute with each other.

5.4.4.6 Soundness Proof for the Alternative Connector

Consider the alternative connector's Petri net semantics as illustrated by Figure 5.24 (on page 135). The proof for this connector is easier than the proof for the parallel connector, as every sequence of transitions of N_{Wc} starting with t_a and ending with t_b either has the form $t_a t_{xors1} \sigma_j t_{xorj1} t_b$, with σ_j being an interleaving of transitions of T_j and $T_x \setminus T_j$ (i.e., the sequence passes through the joinpoint, and not through the advice), or has the form $t_a t_{xors2} \sigma_a t_{xorj2} t_b$, with σ_a being an interleaving of transitions of T_a and $T_x \setminus T_j$ (i.e., the sequence passes through the advice, and not through the joinpoint). For the former case, the proof is trivial, as all firing sequences are equal to the base workflow's firing sequences, with the minor exception of each t_a being replaced by $t_a t_{xors1}$ and each t_b being replaced by $t_{xorj1} t_b$. The latter case corresponds to the application of a replace connector, for which we have already proven soundness in Section 5.4.4.3.

5.4.4.7 Soundness Proof for the Iterating Connector

Consider the iterating connector's Petri net semantics as illustrated by Figure 5.25 (on page 136). In terms of the possible firing sequences of N_{Wc}, this connector does not differ much from the cases already discussed in the previous sections. Suppose a given firing sequence passes through N_j only once, i.e., it does not pass through N_{Wa}. This case is similar to an alternative connector in which the joinpoint branch is chosen instead of the advice branch. Suppose a given firing sequence passes through N_{Wa} $n \geq 1$ times. This case is similar to n subsequent applications of an after connector that introduces N_{Wa} followed by N_j after the N_j that is already present in the base workflow.

5.4.4.8 Soundness Proof for the Synchronizing Connector

Consider the synchronizing connector's Petri net semantics as illustrated by Figure 5.26 (on page 137). Similar to the proofs for the replace and parallel connectors, the proof will benefit from the fact that the joinpoints and the advice are single-entry single-exit fragments, and each of these nets' transitions thus commute with the transitions of the other nets. Remember that the replace connector essentially extracted all the firing sequences of its joinpoint and inserted a valid firing sequence of the advice after each occurrence of t_a, and the parallel connector did not extract any firing sequences but inserted t_{ands}, a valid firing sequence of the advice, and t_{andj} at the appropriate locations between t_a and t_b. Similar to the proof for the parallel connector, we will now introduce t_{ands} before each t_a, t_{andj} after each t_b, and a valid firing sequence of the advice anywhere between the inserted t_{ands} and t_{andj}. Because the two branches are executed in parallel, it is possible that a given firing sequence of the base workflow passes through t_b before passing through t_a. However, because of the commutation mentioned earlier in this paragraph, we can rearrange the transitions of both branches relative to each other in order to prevent this, thus obtaining a valid firing sequence for the composed workflow.

5.4.4.9 Soundness Proof for the Switching Connector

Consider the switching connector's Petri net semantics as illustrated by Figure 5.27 (on page 139). Similar to what we observed when comparing the proof for the alternative connector to the proof for the parallel connector, the proof for the switching connector is easier than the proof for the synchronizing connector, as only one choice can be made at each XOR-split at a time. We thus have the following four cases to consider:

1. We are in a branch in which neither the joining joinpoint nor the splitting joinpoint is located. In this case, there is no difference to the situation in which the connector would not have been applied to the composition.

2. We are in the branch in which the joining joinpoint is located (i.e., the bottom branch of Figure 5.27). Except for the firing of transition t_{xorj2}, there is no difference to the situation in which the connector would not have been applied to the composition.[7]

3. We are in the branch in which the splitting joinpoint is located (i.e., the top branch of Figure 5.27). If transition t_{xors1} is fired, there is no difference to the situation in which the connector would not have been applied to the composition.[8]

4. We are in the branch in which the splitting joinpoint is located. If transition t_{xors2} is fired, the token is removed from the branch and placed in the advice, which we assume to be sound. The token must thus end up in place p'_e. The firing of transition t_{xorj1} moves the token to place p'''_s, and we are in the same situation as in case 2 after the firing of transition t_{xorj2}.[9]

This concludes our presentation of an outline of a proof for each of UNIFY's connectors.

5.5 Summary

The correctness of workflows is essential to workflow management systems and business process management systems. Nevertheless, process designers tend to make many errors. Typical errors are deadlocks (a case gets stuck), livelocks (a case cannot progress), and other anomalies. Repairing such errors can be time consuming and costly. Therefore, workflow verification is highly relevant (van der Aalst et al., 2011). In this chapter, we provide a formalization of UNIFY that is compatible with existing research on this topic within the workflow community, but also addresses the specific notion of connection patterns introduced by UNIFY. In order to formalize the aspect-oriented workflow

[7] In this case, transition t_{xorj1} cannot be fired, because that would have required a different branch to be chosen at the start of the control structure (cf. place p_s and its outgoing transitions).

[8] In this case, transition t_{xors2} cannot be fired, because transition t_{xors1} has removed the token from place p''_e.

[9] In this case, transition t_{xorj2} cannot be fired, because that would have required a different branch to be chosen at the start of the control structure.

concepts introduced by UNIFY, we employ two complementary formalisms. First, we augment the static description of UNIFY's workflows as provided by its base language and connector language meta-models with a static semantics for the weaving of UNIFY connectors using the Graph Transformation formalism. This facilitates static reasoning over the applicability and effects of connectors, and can be used to implement a static weaver of UNIFY connectors. Second, we provide a semantics for the operational properties of workflows by defining a translation to Petri nets, and subsequently extend this semantics to support the operational effects of connectors. This allows reasoning on the dynamics of UNIFY workflow compositions, and can be used to implement a dedicated workflow engine for UNIFY.

In conclusion of this chapter, let us briefly enumerate its contributions:

- We formalize UNIFY's base language meta-model as a *type graph* (Section 5.3.1), and formalize each of UNIFY's connectors as a *graph transformation rule* (Section 5.3.2).
- We use the state-of-the-art Graph Transformation analysis tool AGG to perform a critical pairs analysis of our graph transformation system, which allows investigating possible mutual exclusions and causal dependencies between rules (Section 5.3.3).
- We formalize UNIFY workflow concerns as *workflow graphs* (Section 5.4.2.1), and provide a formal definition of a given workflow graph's corresponding Petri net (Section 5.4.2.2). This definition is compatible with the Petri net patterns for workflow primitives proposed by van der Aalst (1998b).
- We implement the translation of UNIFY workflows to Petri nets as part of the UNIFY implementation, and allow exporting the resulting Petri nets as PNML files supported by existing Petri net editors and analysis tools (Section 5.4.2.2).
- We formalize UNIFY connectors as compositions of the connected concerns' corresponding Petri nets (Section 5.4.3).
- We identify three main properties of our Petri net compositions, and provide an outline for proving the *classical soundness* of a composition, assuming classical soundness of the composed Petri nets (Section 5.4.4).

Chapter 6

Modularizing Workflow Concerns using Concern-Specific Languages

In this chapter, we describe how concern-specific languages *(CSLs) can be built on top of the* UNIFY *base language and connector mechanism which we presented in Chapter 4. We motivate and introduce the notion of concern-specific languages, which has its origins in the notion of domain-specific languages. We propose a general methodology for building CSLs on top of the* UNIFY *approach, and illustrate this methodology by introducing the Access Control and Parental Control CSLs, respectively.*

6.1 Motivation

With PADUS and UNIFY, we have already introduced two approaches that promote separation of concerns in workflow languages by offering an improved modularization mechanism. However, the abstractions offered by existing modularization approaches for workflows, including PADUS and UNIFY, typically remain at the same level as the base workflow: concerns are implemented using the constructs of the base workflow language, which may not be ideally suited to expressing the concern in question in an efficient, elegant or natural way. Although aspect-oriented extensions improve separation of concerns, they introduce an additional layer of complexity that must be bridged in order to communicate about a workflow with domain experts. For example, consider the independently specified workflow concerns of the order handling workflow in Figure 6.1, which we have already used in Chapter 4. In order to augment these workflow concerns with an access control concern that specifies that only premium customers are allowed to specify options using the `SpecifyOptions` activity of the `OrderHandling` concern, one needs to introduce a new XOR-split and XOR-join around the activity, with conditions that query the data perspective of the workflow in order to find out whether the currently logged in user is a premium customer or not. This requires detailed knowledge

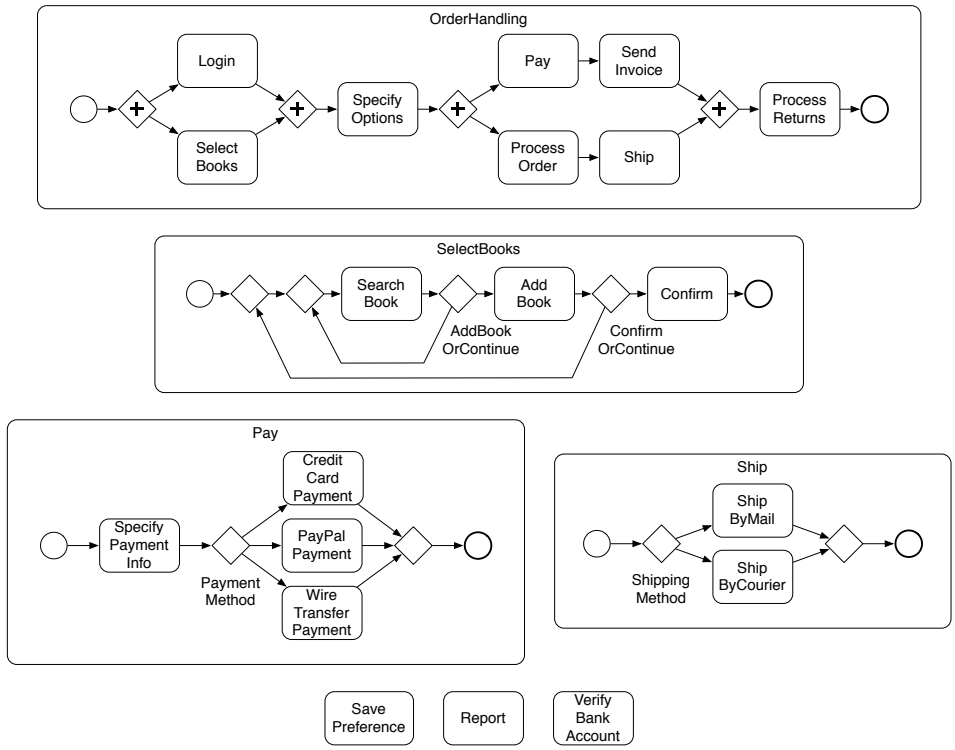


Figure 6.1: Independently specified workflow concerns

of how users are represented in the workflow's implementation. Inspired by the benefits of domain-specific languages in general software engineering (van Deursen et al., 2000), we believe that a means of expressing workflow concerns using abstractions that are closer to the concerns' domains can facilitate expressing workflow concerns, and can improve communication with domain experts. The goal of this chapter is to describe how this is accomplished by building *concern-specific languages* on top of UNIFY.

6.2 From Domain-Specific to Concern-Specific Languages

Domain-specific languages (DSLs) have conceptual roots in the terms *domain languages* (Neighbors, 1980) and *little languages* (Bentley, 1986). A domain language focuses on introducing the concepts and operation of its domain as constructs of a declarative language that can be processed further (Neighbors, 1980). A little language is focused on making programming simple by providing a clean, user-centered syntax and a restricted set of commands as compared to complete programming languages (Bentley, 1986). A DSL is the combination of both concepts: it provides the domain-specific abstractions and notations (Heering, 2000) that are based on the key concepts of its domain (Thibault

et al., 1997). From a technical viewpoint, DSLs can be either external, meaning that their syntax and semantics are defined freely, or internal, meaning that they are embedded in an existing host language (Günther, 2010). Using DSLs in application development increases development productivity by efficient code reuse and a reduction of errors (Czarnecki and Eisenecker, 2000; Greenfield et al., 2004).

Concern-specific languages (CSLs; cf. Bodden, 2005; Braem et al., 2006a) share the goals of DSLs. A CSL facilitates separation of concerns by offering notations and abstractions for expressing a family of concerns; typically, this family of concerns will be *crosscutting*. The earliest aspect-oriented programming languages were each developed for a particular family of crosscutting concerns. Examples of these early CSLs are COOL (Lopes and Kiczales, 1997; Lopes, 1997), a language for expressing the aspect of synchronization for programs written in Java, and RIDL (Lopes and Kiczales, 1997; Lopes, 1997), a language for expressing the aspect of data serializability in distributed environments. A more recent concern-specific language is KALA (Fabry, 2005), a language for describing the use of advanced transaction models by an application, which also allows new models to be defined if needed. The use of aspect-oriented technology to implement CSLs makes it especially easy to implicitly or explicitly quantify over events in the application's control flow (Bodden, 2005). In previous work (Braem et al., 2006a), we have introduced a CSL for a family of concerns related to *billing* within web service orchestrations. By defining a billing concern in a module-like entity, it becomes simpler to apply this concern to several events within an orchestration without compromising its maintainability. Inspired by our earlier research on enabling the definition of concern-specific languages on top of PADUS (Braem et al., 2006a,c), our current objective is to enable the definition of concern-specific languages on top of the UNIFY framework.

We introduce the concept of CSLs to the UNIFY framework in order to facilitate the definition and application of concerns. In Section 6.3, we propose a general methodology for building CSLs on top of the UNIFY approach. In Sections 6.4 and 6.5, we illustrate this methodology by introducing the Access Control and Parental Control CSLs, respectively.

6.3 General Methodology

In this section, we first describe the different actors involved in the development of a CSL, and subsequently describe each of the three steps of our methodology.

Actors In general, three kinds of actors are involved in the development of a CSL:

- **Domain experts.** One of the main goals of adding CSLs on top of UNIFY is to include domain experts in the workflow development process.[1] Thus, domain experts will be an important class of users of CSLs. Domain experts are also included in the development of the actual CSLs, as their domain expertise is essential to identifying the correct domain concepts and appropriate syntax for the CSLs.

[1] Whenever we use the term "domain" in the context of a CSL, we mean the domain of the CSL's family of concerns.

- **Workflow developers.** Although domain experts are the most important class of users of CSLs, workflow developers may use CSLs as well because of their superior suitability for expressing the concerns at hand. Workflow developers are also included in the development of the actual CSLs, as their knowledge of the concrete workflows used within the organization is essential to correctly translating domain concepts to workflow concepts.
- **CSL developers.** In addition to domain experts and workflow developers, an additional kind of actor is necessary to actually develop a new CSL before it can be used: we expect CSLs to be developed by IT specialists who are familiar with both the UNIFY API and the notion of concern-specific languages, based on discussions with domain experts and workflow developers. Once a CSL has been developed, both domain experts and workflow developers should be able to specify concerns using CSL artifacts without further intervention of these CSL developers.

Step 1: Identifying domain concepts and relations The first step in developing a CSL is identifying the relevant domain concepts and relations between these domain concepts. Because the CSL developer typically has insufficient knowledge of the domain, domain experts will provide valuable input for this step. This step will typically result in a UML class diagram, which will be used during the following steps.

Step 2: Specifying a syntax for CSL artifacts The second step in developing a CSL is specifying an appropriate syntax that matches the concepts and relations identified in the previous step. Domain experts can be included in this step in order to ensure the syntax meets their expectations. The kind of syntax (textual, XML, visual, ...) determines how it will be specified. For example, a textual syntax can be specified using BNF, an XML syntax using XML SCHEMA, etc.

Step 3: Translating CSL artifacts to UNIFY artifacts The third step in developing a CSL is providing a mechanism by means of which artifacts expressed using the above syntax are translated into basic UNIFY artifacts such as CompositeActivities and Connectors. This can be accomplished by implementing a preprocessor for CSL artifacts. During this step, workflow developers will provide valuable input on how domain concepts are to be translated to UNIFY workflow concepts.

In Sections 6.4 and 6.5, we introduce the Access Control and Parental Control CSLs, respectively. For each of these CSLs, we follow the above methodology. We have used a graphics editor to specify the UML class diagrams that define the domain concepts and relations of the CSLs. Each of the CSLs has an XML syntax, which is specified using XML SCHEMA. Each of the CSLs' artifacts is translated to UNIFY artifacts using a preprocessor that has been implemented from scratch. This encompassed writing a parser for the CSL's artifacts that returns a JAVA representation of the CSL artifacts' domain concepts, and writing a code generator that generates the appropriate UNIFY CompositeActivity and Connector for each CSL artifact using the JAVA representation of its domain concepts. Günther et al. (2012) have identified a number of design dimensions that are relevant to

the design of DSLs, i.e., (1) *syntax*, (2) *language base*, (3) *artifact type*, and (4) *artifact abstraction*. Along these dimensions, the CSLs we develop (1) use *textual notations*, (2) are *external DSLs*, (3) use *code artifacts*, and (4) use *internally structured* artifacts, respectively.

It is feasible to employ the above approach to develop new CSLs. However, inspired by current tool support for development of domain-specific languages (such as Xtext[2] and JetBrains MPS[3]), we envision that development of a CSL can be facilitated using tool support as well. Such a CSL development tool would include a UML editor to model the domain concepts and relations of a new CSL. The tool would then support creating a CSL syntax in a user-friendly way based on this UML model. By allowing to map CSL concepts to UNIFY workflow concepts, the tool would support semi-automatically generating a preprocessor that translates CSL artifacts to UNIFY artifacts. Note that different concerns expressed using the same CSL will typically have a similar structure. Therefore, one can conceive schemes that allow specifying concerns based on concern-specific *templates*. Such templates can be specified after defining the CSL syntax, and the definition of such templates would thus be integrated in the CSL development tool, while their use would be integrated in a CSL artifact specification tool. This tool support for CSL development is subject to future work. Therefore, we now proceed with the introduction of our Access Control CSL.

6.4 The Access Control CSL

Access control deals with specifying which operations can be performed by which users. For example, one might want to specify that the execution of some workflow activities is only allowed for certain users. In this section, we define an Access Control CSL that accomplishes this. Our Access Control CSL is inspired by *flat role-based access control* (Sandhu et al., 2000), which embodies the essential aspects of role-based access control (RBAC). The basic concepts of RBAC are *users*, *roles*, and *permissions*. A user may be associated with any number of roles, and a role may be associated with any number of permissions. Thus, users acquire permissions by being associated with roles.

6.4.1 Language

The first step in developing a CSL is to identify the relevant domain concepts based on discussions with domain experts. Figure 6.2 shows the domain concepts of our Access Control CSL, as well as the relations between these concepts. At the top left of the figure, we see that a User may belong to any number of Roles, and that a Role may group any number of Users. Likewise, a Role may group any number of Permissions, and a Permission may belong to any number of Roles. Every Permission refers to a single Operation, but an Operation may be referred to by any number of Permissions. A Permission is either an AllowPermission, i.e., a permission that specifies that users with the given role may perform the given operation, or a DenyPermission, i.e., a permission that specifies that

[2] Cf. `http://www.eclipse.org/Xtext/`
[3] Cf. `http://www.jetbrains.com/mps/`

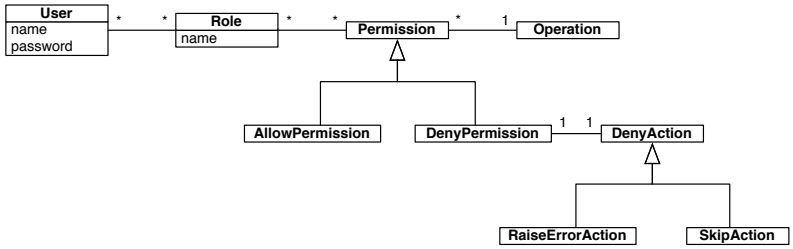


Figure 6.2: Access Control CSL domain concepts and relations

users with the given role may *not* perform the given operation. Every DenyPermission has a one-to-one association with a DenyAction, which specifies what should happen when a user with the given role attempts to perform the given operation. This is either raising an error (RaiseErrorAction) or skipping the operation (SkipAction). In the context of workflows, the notion of a user is typically present in a workflow's data perspective, and operations correspond to the execution of workflow activities.

The second step in developing a CSL is to map the previously identified domain concepts to a syntax that is appropriate — based on discussions with domain experts — to specifying the concern at hand. For our Access Control CSL, we have opted for an XML syntax, as we believe that of all textual syntaxes, XML will be most familiar to non-programmers such as domain experts. Additionally, the use of XML is very common in traditional workflow development, as most current workflow languages such as WS-BPEL and BPMN have an XML representation. We have defined an XML syntax for our Access Control CSL that closely matches the concepts and relations we identified above. Listing 6.1 provides an example concern that is expressed using this syntax. The example specifies how the access to the `SpecifyOptions` activity of the `OrderHandling` concern in Figure 6.1 should be controlled. An access control concern is specified using the `<AccessControlConcern>` element (cf. line 1). First, a default permission is specified, which is `Allow` in this example (cf. line 2). Next, a number of roles are specified using `<Role>` elements (cf. lines 3–6). Each role may contain any number of permissions, which are specified using the `<Allow>` and `<Deny>` elements (cf. line 5). Because the default permission in this example is `Allow`, we only need to explicitly list the activities to which access should be denied. Next, a number of users are specified using `<User>` elements (cf. lines 7–12). Each user is assigned to one or more roles. Finally, the `<DefaultUser>` element (cf. lines 13–15) specifies the roles of the default user, and the `<UsernameVariable>` element (cf. line 16) specifies in what variable of the workflow the name of the currently logged in user can be found.[4] The full XML syntax of our Access Control CSL is defined using XML SCHEMA, and is provided in Appendix C.

When we apply an access control concern to a workflow, new behavior will be executed around each of the activities for which a "Deny" permission is specified. In our example, this would be the `SpecifyOptions` activity. This new behavior will check

[4] We discuss the relation between concern-specific languages and the data perspective in Section 6.6.

```
1 <AccessControlConcern name="Example1">
2   <DefaultPermission permission="Allow" />
3   <Role name="PremiumCustomer" />
4   <Role name="NormalCustomer">
5     <Deny activity="SpecifyOptions" action="Skip" />
6   </Role>
7   <User name="john">
8     <UserRole role="PremiumCustomer" />
9   </User>
10  <User name="mike">
11    <UserRole role="NormalCustomer" />
12  </User>
13  <DefaultUser>
14    <UserRole role="NormalCustomer" />
15  </DefaultUser>
16  <UsernameVariable name="username" />
17 </AccessControlConcern>
```

Listing 6.1: Example access control concern

whether the currently logged in user is allowed to execute the activity in question, and will either raise an error or skip the activity if this is not the case. In our example, this means that only premium customers (i.e., `john`) would be able to execute the `SpecifyOptions` activity, whereas the activity would be skipped for all other customers. All of this is accomplished by generating UNIFY artifacts based on the CSL artifacts, as is explained in Section 6.4.2.

6.4.2 Translation to UNIFY

The third and final step in developing a CSL is to implement a preprocessor that translates the CSL's artifacts into UNIFY artifacts, based on discussions with workflow developers. The constructs offered by UNIFY are well suited to implementing CSLs such as those we introduce in this chapter. For our Access Control CSL, the translation is performed as follows. When translating a given access control concern, the translation process makes a distinction between activities for which at least one "Deny" permission is specified, and activities for which no "Deny" permission is specified.

Activities for which at least one "Deny" permission is specified Depending on the permission of the currently logged in user, such activities may need to be skipped, or the attempt to use such an activity may raise an error. An *around connector* (cf. Section 4.5.3.1) is generated with a pointcut that selects each of the activities as joinpoints (cf. `GeneratedAccessControlConnector` in Listing 6.2). The connector's advice is a new CompositeActivity, which is shown as a BPMN diagram in Figure 6.3. The corresponding WS-BPEL code is listed in Listing 7.7 on page 180. The CompositeActivity's behavior is as follows:

```
GeneratedAccessControlConnector:
CONNECT GeneratedAccessControlActivity
AROUND activity("OrderHandling\.SpecifyOptions")
PROCEEDING AT
    activity("GeneratedAccessControlActivity\.GeneratedDummyActivity")
```

Listing 6.2: Generated around connector for the example access control concern

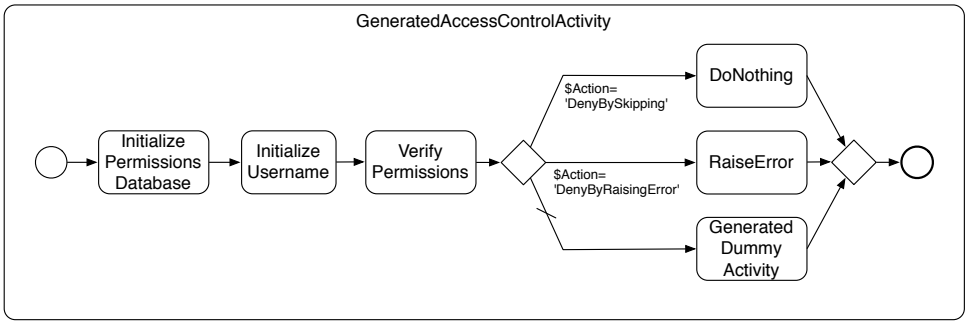


Figure 6.3: Generated composite activity for the example access control concern

1. The `InitializePermissionsDatabase` activity initializes a variable with a mapping from activities and usernames to permissions, e.g., *("SpecifyOptions", "mike")* → *"DenyBySkipping"*. This mapping is generated by reasoning about the User–Role and Role–Permission relations of the access control concern.

2. The `InitializeUsername` activity initializes a variable with the value of the username variable, whose name is specified in the access control concern. This is the variable that contains the name of the currently logged in user.

3. The `VerifyPermissions` activity determines the action to performed for the currently logged in user by querying the variables initialized in the previous steps.

4. An XOR-split splits the control flow into three branches based on the action determined in the previous step.

   - If the action to be performed is *"DenyBySkipping"*, control flow is directed to a branch containing an activity that does nothing.
   - If the action to be performed is *"DenyByRaisingError"*, control flow is directed to a branch containing an activity that raises an error.
   - Otherwise, control flow is directed to a branch containing a dummy activity. The connector specifies that this activity is the advice's *proceed* activity.

5. An XOR-join joins the above branches.

Finally, the generated UNIFY connector and CompositeActivity are applied to the workflow, so that they can be woven as described in Section 4.5.

Activities for which no "Deny" permission is specified Such activities do not necessarily need any additional behavior inserted around it, as the activity would never need to be skipped, or the attempt to use it would never need to cause the raising of an error. Therefore, we do not generate any UNIFY artifacts in this case. Alternatively, one could employ the same connector and CompositeActivity as in the previous case; this would simply mean that the proceed branch of the advice would be followed every time.

Based on the above, we can conclude that the around connector is sufficient for implementing our Access Control CSL in terms of UNIFY. As we explain in the following section, the Parental Control CSL will require other, more advanced connectors.

6.5 The Parental Control CSL

Parental control deals with ensuring that children can only access content or perform operations that are appropriate to their age. Unlike access control, which is a topic of significant interest in areas such as computer security and domain-specific languages, parental control is not a common topic in computer science research. We thus cannot base our concern-specific language for parental control on scientific articles, but rather use information available on the world wide web.[5] In general, one can identify three main kinds of parental control: *filtering* allows restricting access to age appropriate content, *usage control* allows restricting the operations that a child may perform, and *monitoring* allows reviewing the operations that are performed by a child.

6.5.1 Language

Figure 6.4 shows the domain concepts of our Parental Control CSL, as well as the relations between these concepts. Remember that these are typically identified based on discussions with domain experts. At the top left of the figure, we see that a Parent may have any number of Children, and a Child may have any number of Parents. At any point in time, a Child has a single Age. Policies are expressed in terms of the Age of Children. Each Policy refers to a single Operation. There are three kinds of Policies: FilteringPolicies, UsagePolicies, and MonitoringPolicies. FilteringPolicies allow restricting Children's access to the given Operation's results, UsagePolicies allow restricting the access to the given Operation itself, and MonitoringPolicies allow monitoring the access to the given Operation. A UsagePolicy is either a DenyUsagePolicy, which simply denies access to the Operation, or a ReferUsagePolicy, which refers the Operation from a Child to one of its Parents (e.g., payment by credit card should be performed by a parent rather than by the child). In the context of workflows, the notions of children and parents correspond to workflow users, and operations correspond to the execution of workflow activities.

[5] For example, Wikipedia's article on "Parental controls", cf. `http://en.wikipedia.org/wiki/Parental_controls`

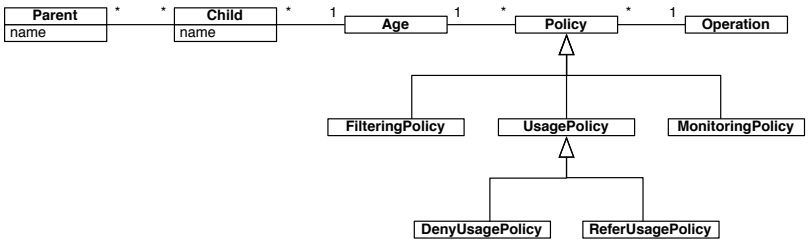


Figure 6.4: Parental Control CSL domain concepts and relations

We have defined an XML syntax for our Parental Control CSL that closely matches the concepts and relations we identified above. Remember that such a syntax is typically defined based on discussions with domain experts. Listing 6.3 provides an example concern that is expressed using this syntax. The example specifies parental control policies for some of the activities of the concerns in Figure 6.1. A parental control concern is specified using the `<ParentalControlConcern>` element (cf. line 1). First, a number of policies are specified. Filtering policies are specified using the `<Filter>` element (cf. lines 3–6), usage policies are specified using the `<Deny>` and `<ReferToParent>` elements (cf. lines 7–8), and monitoring policies are specified using the `<Monitor>` element (cf. line 9). Policies are grouped using one or more `<Policies>` elements (cf. lines 2–10), which allow specifying the age of the children to which the policies apply. Any number of `<Child>` elements (cf. lines 11–13) may be used to specify a number of children and their parents, and the `<AgeVariable>` element (cf. line 14) specifies in what variable of the workflow the age of the currently logged in user can be found. The full XML syntax of our Parental Control CSL is defined using XML SCHEMA, and is provided in Appendix C.

When a parental control concern is applied to a workflow, new behavior will be executed around each of the activities for which a policy is specified. In our example, this would be the `SearchBook`, `SpecifyOptions`, `SpecifyPaymentInfo`, and `Confirm` activities. The actual new behavior depends on the kind of policy: the `SearchBook` activity's results will thus be filtered according to the exclude declarations if the current user is younger than 18; the execution of the `SpecifyOptions` activity will be denied (i.e., the activity will be skipped) if the current user is younger than 18; the execution of the `SpecifyPaymentInfo` activity will be referred to the current user's parent if he/she is younger than 18; and the execution of the `Confirm` activity will be monitored if the current user is younger than 18. All of this is accomplished by generating UNIFY artifacts based on the CSL artifacts, as is explained in Section 6.5.2.

```
<ParentalControlConcern name="Example2">
  <Policies youngerThan="18">
    <Filter activity="SearchBook" resultVariable="books">
      <Exclude property="Rating" value="Mature" />
      <Exclude property="Genre" value="Horror" />
    </Filter>
    <Deny activity="SpecifyOptions" />
    <ReferToParent activity="SpecifyPaymentInfo"
        usernameVariable="username" />
    <Monitor activity="Confirm" usernameVariable="username" />
  </Policies>
  <Child name="suzy">
    <Parent name="george" />
  </Child>
  <AgeVariable name="age" />
</ParentalControlConcern>
```

Listing 6.3: Example parental control concern

CSL artifact	**UNIFY connector**
Access control concern	*Around* connector that applies to all activities for which a *Deny* permission is specified
Parental control concern — Filtering	*After* connector that applies to all of the policy's activities
Parental control concern — Deny usage	*Alternative* connector that applies to all of the policy's activities
Parental control concern — Refer usage	*Alternative* connector that applies to all of the policy's activities
Parental control concern — Monitoring	*Parallel* connector that applies to all of the policy's activities

Table 6.1: Mapping from CSL artifacts to UNIFY connectors

6.5.2 Translation to UNIFY

Table 6.1 gives an overview of the UNIFY connectors to which parental control concerns are translated.[6] Remember that this translation is typically based on discussions with workflow developers. The translation from parental control concerns to UNIFY is performed as follows. When translating a given parental control concern, the translation process makes a distinction between the different policies defined by the concern.

Activities for which a filtering policy is specified For each filtering policy, an *after connector* (cf. Section 4.5.3.1) is generated with a pointcut that selects the activity whose results need to be filtered (cf. `GeneratedFilteringConnector` in Listing 6.4). The con-

[6] For comparison, we also include the UNIFY connectors to which access control concerns are translated (cf. Section 6.4.2).

```
GeneratedFilteringConnector:
CONNECT GeneratedFilteringActivity
AFTER activity("SelectBooks\.SearchBook")

GeneratedDenyUsageConnector:
CONNECT GeneratedDenyUsageActivity
ALTERNATIVE TO activity("OrderHandling\.SpecifyOptions")
IF "$age < 18"

GeneratedReferUsageConnector:
CONNECT GeneratedReferUsageActivity
ALTERNATIVE TO activity("Pay\.SpecifyPaymentInfo")
IF "$age < 18"

GeneratedMonitoringConnector:
CONNECT GeneratedMonitoringActivity
PARALLEL TO activity("SelectBooks\.Confirm")
```

Listing 6.4: Generated after, alternative, and parallel connectors for the example parental control concern

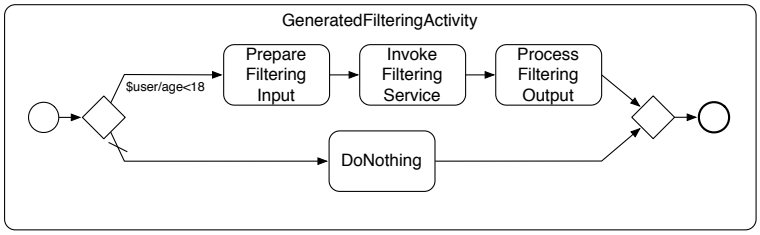


Figure 6.5: Generated composite activity for the example parental control concern's filtering policy

nector's advice is a new CompositeActivity, which is shown as a BPMN diagram in Figure 6.5. The corresponding WS-BPEL code is listed in Section D.1. The CompositeActivity starts with an XOR-split. One branch is followed when the value of the age variable is lower than the policy's age limit. This branch performs the filtering by invoking an external service: a first activity prepares the input for the external service by copying the contents of the policy's result variable and the exclude statements to the service invocation's input variable, a second activity actually invokes the service, and a third activity copies the service's output back to the policy's result variable. The XOR-split's second branch does nothing. Both branches are joined by an XOR-join.

Activities for which a deny usage policy is specified For each deny usage policy, an *alternative connector* (cf. Section 4.5.3.1) is generated with a pointcut that selects the activity whose results need to be denied (cf. `GeneratedDenyUsageConnector` in List-

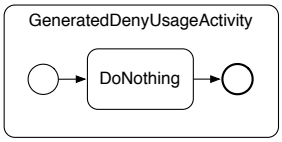


Figure 6.6: Generated composite activity for the example parental control concern's deny usage policy

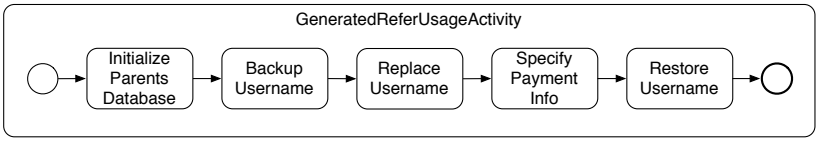


Figure 6.7: Generated composite activity for the example parental control concern's refer usage policy

ing 6.4). The connector's condition specifies that the advice is executed when the value of the age variable is lower than the policy's age limit. The connector's advice is a new CompositeActivity, which is shown as a BPMN diagram in Figure 6.6. The corresponding WS-BPEL code is listed in Section D.2. The CompositeActivity merely does nothing. Thus, the joinpoint activity will be skipped if the currently logged in user's age is lower than the policy's age limit.

Activities for which a refer usage policy is specified For each refer usage policy, an *alternative connector* (cf. Section 4.5.3.1) is generated with a pointcut that selects the activity whose results need to be referred (cf. `GeneratedReferUsageConnector` in Listing 6.4). The connector's condition specifies that the advice is executed when the value of the age variable is lower than the policy's age limit. The connector's advice is a new CompositeActivity, which is shown as a BPMN diagram in Figure 6.7. The corresponding WS-BPEL code is listed in Section D.3. The CompositeActivity's behavior is as follows. First, a variable is initialized with a mapping from children to their parents. Second, the current value of the username variable is saved in a new, temporary variable. Third, the username variable is assigned with the name of the currently logged in user's parent. Fourth, a copy of the joinpoint activity is executed, and finally, the username variable is assigned with its old value that was saved in the temporary variable. Because the username variable of the joinpoint activity is changed, the joinpoint activity will now refer to the parent instead of the child.

Activities for which a monitoring policy is specified For each monitoring policy, a *parallel connector* (cf. Section 4.5.3.1) is generated that selects the activity that needs to be monitored (cf. `GeneratedMonitoringConnector` in Listing 6.4). The connector's advice is a new CompositeActivity, which is shown as a BPMN diagram in Figure 6.8. The corresponding WS-BPEL code is listed in Section D.4. The CompositeActivity starts with

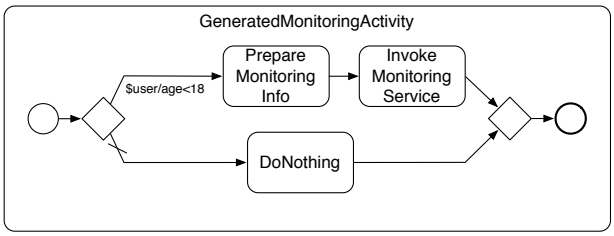


Figure 6.8: Generated composite activity for the example parental control concern's monitoring policy

an XOR-split. One branch, which is followed when the value of the age variable is lower than the policy's age limit, consists of an activity that prepares the monitoring information (the name of the joinpoint activity and the name of the currently logged in user), and an activity that invokes a monitoring service with the monitoring information as input. A second branch, which is followed in all other cases, consists of an activity that simply does nothing. Both branches are joined by an XOR-join.

This concludes the development of our example CSLs. We will now discuss a number of important issues regarding our concern-specific layer.

6.6 Discussion

Focus of our research The main focus of our research is the improvement of modularity in workflows. Thus, we investigate the suitability of concern-specific languages for this purpose. We are explicitly not focusing on how one can create the best concern-specific language for a given concern, as this would lead us too far into the research area of domain-specific languages. We do not claim that the example CSLs we develop in this chapter are the best CSLs for workflows, or the most typical ones; they are merely well suited for the running example we use throughout this chapter. Other CSLs may be appropriate in other workflows. In the past, we have researched the *billing* concern in web service orchestrations, and have made an initial attempt at developing a concern-specific language for it (Braem et al., 2006c,a). A similar CSL can be developed on top of UNIFY.

Benefits of CSLs to workflow development We observe the following benefits when applying concern-specific languages to workflow development:

- Concerns are defined at an abstraction level that is closer to the domain of the concern. Thus, **less technical knowledge** is required to specify concerns, which allows including domain experts in specifying the concerns. This may facilitate the implementation of workflows that conform to business requirements. For example, the access control concern of Listing 6.1 refers mostly to domain concepts,

whereas its corresponding UNIFY implementation refers only to general workflow concepts such as activities and data (cf. Listing 7.7 on page 180 for the actual code, and Section 7.2.1 for its discussion).

- Concern-specific languages hide the complexity of the underlying implementation by adopting a more declarative approach. Because the different concerns expressed using a given CSL by definition have much in common, this commonality can be implemented in advance in the CSL infrastructure, and only the variability must be specified by the users of the CSL in the CSL's artifacts. Thus, **less code** is required to specify concerns. For example, the access control concern of Listing 6.1 is specified in 17 lines of code using the Access Control CSL, compared to 67 lines of code using standard UNIFY (cf. Listing 7.7 on page 180 for the actual code, and Section 7.2.1 for its discussion).

These benefits are currently only validated by our initial case studies, but are consistent with findings in more general research on domain-specific languages (van Deursen et al., 2000). A more extensive investigation of these and other issues (such as usability) based on a real-life case study would provide more conclusive results.

Benefits of UNIFY to CSL implementation Concern-specific languages have originated in the research domain of aspect-oriented programming, which focuses on modularizing crosscutting concerns. The workflow concerns for which we introduced CSLs in Chapter 6 are typically crosscutting, too. Thus, the implementation of the CSLs is facilitated when the underlying technology supports applying behavior at different places in a workflow, which is the case in UNIFY due to its ExternalConnectors. The novel connectors offered by UNIFY, such as the *parallel* and *alternative* connectors, allow implementing the concerns in a way that is less cumbersome than aspect-oriented approaches that only offer the traditional *before*, *after* and *around* advices. We compare UNIFY's novel connectors with existing approaches in more detail in Section 7.2.2.

Concern-specific languages and the data perspective The CSLs we present in Chapter 6 abstract over the concrete control flow details of the expressed concerns: with regard to the control flow perspective, the persons who specify concerns using CSLs merely need to decide what activities in the base workflow the concerns should be applied to. Given the prevalence of workflow models in business process management, it seems feasible that domain experts perform this task. With regard to the data perspective, however, this may not be the case: our current CSLs reference variables of the base workflow, which are not necessarily modeled in the abstract workflow models with which domain experts are familiar. This issue can be tackled by providing tool support for specifying concerns using CSLs. For example, a tool could provide an overview of the base workflow's data model, and allow viewing the list of variables used by each of the workflow's activities, perhaps showing some additional documentation of the base workflow's data perspective that was prepared by workflow developers. A more conceptual approach could comprise a mapping from the actual base workflow's data model to a higher-level

representation of data in terms of domain concepts.[7] This, however, is beyond the scope of our current approach.

6.7 Summary

Because workflows are well suited to representing processes, e.g., in a visual form, they are often used for communicating with *domain experts*. However, domain experts typically do not develop workflows by themselves, as specific technical knowledge is required for augmenting high-level process descriptions with low-level implementation details. *Workflow developers* fulfill this latter role and thus perform most of the workflow development. In this chapter, we aim to bridge the gap between the domain level and the implementation level of workflows in order to facilitate the definition of workflow concerns and improve communication with domain experts. This is accomplished by building *concern-specific languages* — languages which are specifically tailored towards expressing a single family of concerns — on top of the UNIFY framework. In order to illustrate how this can be done, we develop two example CSLs, one for the family of access control concerns, and one for the family of parental control concerns. We propose a methodology for developing concern-specific languages, which consists of three main steps: (1) identifying domain concepts and relations, (2) specifying a syntax for CSL artifacts, and (3) translating CSL artifacts to UNIFY artifacts. Finally, we outline how this methodology can be further supported by additional tool support.

[7] For an approach that addresses similar issues in the context of business rules for object-oriented applications, see the work by Cibrán (2007).

Chapter 7

Implementation and Validation of UNIFY

In this chapter, we provide an overview of UNIFY*'s implementation, and subsequently present a validation of the approach with respect to expressiveness, performance, and scalability.*

7.1 Implementation

We have created a proof-of-concept implementation for UNIFY, which is available for download from the UNIFY website (Joncheere et al., 2012). The general architecture of this implementation is shown in Figure 7.1. At the heart of the architecture lies a JAVA implementation of the UNIFY base language (cf. Figures 4.2 and 4.4) and connector mechanism (cf. Figure 4.14). The UNIFY API allows constructing and manipulating composite activities in-memory, while instantiations of the UNIFY framework provide parsers and serializers for existing concrete workflow languages, i.e., WS-BPEL and BPMN. A composition specifies which composite activities are to be loaded and which connectors are to be applied to them. One by one, the UNIFY weaver applies the connectors to the base workflow (which is a composite activity) in the order specified by the composition. For each connector, the base workflow is modified accordingly. The final woven composite activity is transformed into a Petri net execution model if one wants to use UNIFY's built-in execution engine, or is exported back to the workflow language in which the original composite activities were specified. The concern-specific layer allows CSL artifacts to be applied to a base workflow. The CSL artifacts are parsed into a JAVA representation of the CSLs' concepts. These are subsequently translated into standard UNIFY connectors and composite activities, which are woven as described above.

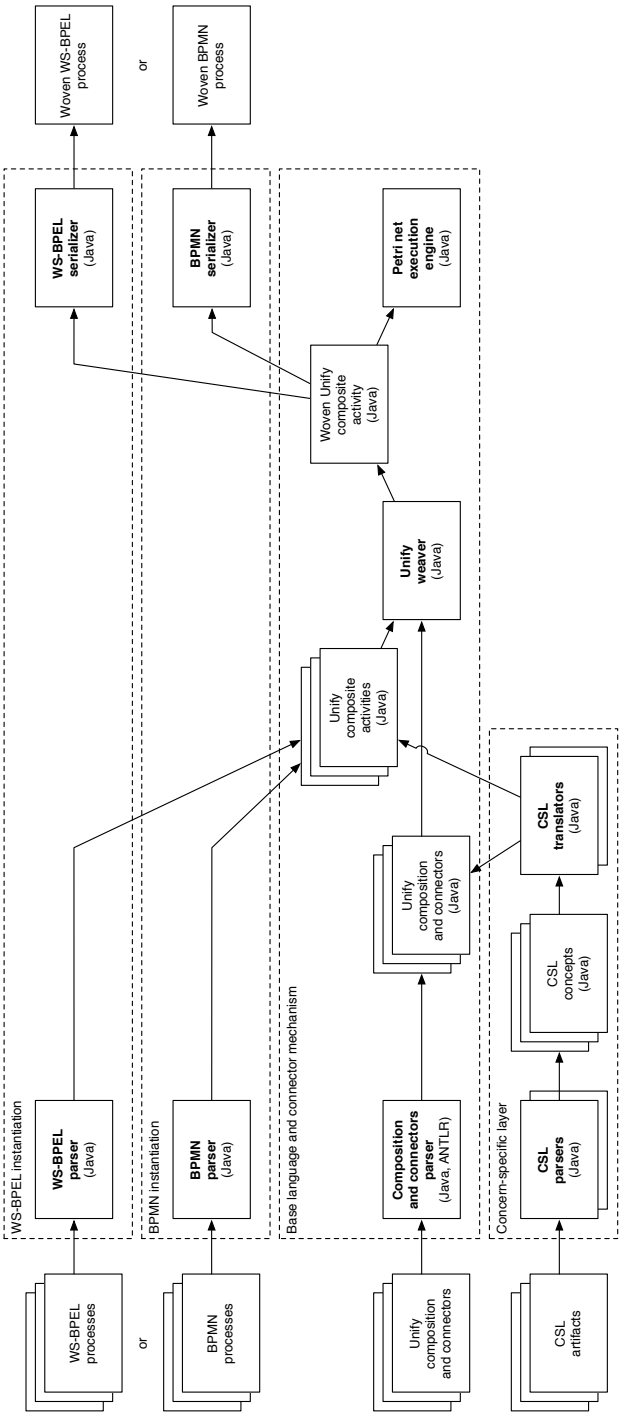


Figure 7.1: General architecture of the UNIFY implementation

7.1.1 JAVA Implementation of the UNIFY Base Language and Connector Mechanism

In Section 4.3, we presented UNIFY's base language meta-model (cf. Figures 4.2 and 4.4), and in Section 4.5, we presented UNIFY's connector language meta-model (cf. Figure 4.14). At the heart of the UNIFY framework implementation lies a JAVA implementation of these meta-models. The base language's interface is implemented using 29 interfaces and a total of 239 lines of code, while a default implementation is provided using 29 classes and a total of 1488 lines of code. The connector language is implemented using 31 classes and a total of 817 lines of code. UNIFY activities (as well as their composition using connectors) can be visualized by supporting the translation of UNIFY activities to GRAPHVIZ files.[1]

7.1.2 Instantiations of the UNIFY Framework

We instantiated the UNIFY framework towards two concrete workflow languages. As a first language, we chose WS-BPEL, which is the de facto standard in executable workflow languages, and can be considered representative of block-structured workflow languages. As a second language, we chose the Business Process Model and Notation (BPMN), which is the de facto standard *notation* for graphically specifying workflows, and has recently been extended with an underlying execution model. BPMN can be considered representative of graph-based workflow languages. The framework is instantiated by implementing the interfaces and/or extending the classes of Section 7.1.1. For WS-BPEL, this is accomplished using 35 classes and a total of 1082 lines of code. For BPMN, this is accomplished using 13 classes and a total of 368 lines of code. Parsers and serializers for each workflow language ensure that existing artifacts can be imported and new artifacts can be exported. The WS-BPEL parser consists of 749 lines of code, the BPMN parser of 217 lines of code, the WS-BPEL serializer of 845 lines of code, and the BPMN serializer of 239 lines of code.

Instantiating UNIFY towards concrete workflow languages is straightforward for common workflow concepts, i.e., activities and basic control flow concepts. However, three limitations may arise when instantiating UNIFY:

1. Workflow languages are typically either graph-based (such as BPMN and YAWL), block-structured (such as the part of WS-BPEL that is most commonly used), or both (such as the complete WS-BPEL language). UNIFY is graph-based: a workflow consist of nodes that are connected to each other using transitions, with control nodes being used to split and join control flow. Block-structured constructs such as the sequential, parallel, conditional and iterating control structures encountered in WS-BPEL can be straightforwardly mapped to UNIFY's graph-based constructs. However, mapping UNIFY's graph-based constructs to block-structured constructs requires the UNIFY workflow to be structured (i.e., every split must have a corresponding join).

[1] Cf. `http://www.graphviz.org`

2. The UNIFY base language only provides the basic control flow patterns identified in existing research (cf. Table 4.2 on page 65). These basic control flow patterns are the ones that are supported by most workflow languages (Russell et al., 2006a). Support for more advanced control flow patterns such as those encountered in YAWL (van der Aalst and ter Hofstede, 2005) is subject to future work. However, we do not foresee any fundamental obstacles to extending UNIFY with support for additional control flow patterns.

3. The UNIFY base language focuses on the control flow and data perspectives. It does not currently address other perspectives, such as the exception handling perspective (Russell et al., 2006b). Support for these perspectives is subject to future work. Again, we do not foresee any fundamental obstacles to extending UNIFY with support for additional perspectives.

These limitations are the result of the deliberate choice to focus on the expressiveness of the modularization mechanism rather than on the expressiveness of the individual modules, and can be addressed by iterating over the base language meta-model. We believe that the current meta-model is sufficient for demonstrating our contributions to the modularization of workflow concerns.

7.1.3 Connectors and Compositions

In Section 4.5, we presented UNIFY's connector mechanism, which introduces the connector and composition constructs. Connectors are defined in plain text files using the syntax discussed in Section 4.5.3 and listed in Appendix A. They are parsed using a parser and tree parser generated using the ANTLR parser generator,[2] based on two ANTLR grammar files of 139 and 237 lines of code, respectively. Compositions are defined in XML files using the syntax discussed in Section 4.5.4, and are parsed using a manually implemented JAXP SAX parser of 133 lines of code.

7.1.4 The UNIFY Weaver

After creating a number of concerns using the JAVA classes of Section 7.1.1 or importing a number of concerns using the WS-BPEL or BPMN parsers of Section 7.1.2, they can be used in the UNIFY connectors and compositions of Section 7.1.3. These concerns, connectors and compositions are the input for the UNIFY weaver, whose goal is to generate a single woven workflow for each input workflow composition. We have implemented this weaver in JAVA. In order to weave a composition, the weaver will process each of the composition's connectors one at a time, according to the order specified in the composition. To this end, we have implemented each of the connectors of Section 4.5 in JAVA: for each connector, the base concern's JAVA representation is modified to include the advice concern at the joinpoints selected by the connector's pointcut. At the end of the weaving process, the modified base concern is returned as the woven workflow. The weaver class consists of 360 lines of code. As part of the instantiation of UNIFY towards WS-BPEL

[2] Cf. `http://www.antlr.org/`

and BPMN, the weaver has been extended to two classes consisting of 282 and 61 lines of code, respectively.

7.1.5 The UNIFY Petri Net Engine

In addition to executing a woven workflow composition using a standard workflow engine, UNIFY supports executing a workflow composition using a custom Petri net based workflow engine, based on the Petri net semantics we proposed in Chapter 5. To this end, we implemented the transformation of a UNIFY workflow to a Petri net model in JAVA, according to the algorithm proposed in Section 5.4.2. The Petri net model is implemented using a total of 14 classes and 541 lines of code, while the transformation is implemented using a visitor of 168 lines of code. The Petri net model can be executed using a total of 3 classes and 221 lines of code (with most of the execution logic being implemented in the Petri net model itself). We have implemented a visitor of 157 lines of code that allows exporting the Petri net model as a PNML file, which is a standard for the exchange of Petri nets supported by existing Petri net editors and analysis tools. Note that the current implementation of our Petri net engine uses the UNIFY weaver to weave the process before transforming it to a Petri net. Extending the Petri net engine with support for run-time weaving is subject to future work, but the semantics proposed in Chapter 5 is an ideal basis for this.

7.1.6 Concern-Specific Languages

We already described a methodology for developing CSLs in Section 6.3. We followed this methodology for both of the CSLs we implemented on top of UNIFY:

- For the **Access Control CSL**, we implemented the domain concepts and relations of Figure 6.2 on page 156 in JAVA. This required a total of 8 classes and 230 lines of code. A parser of 116 lines of code is used to parse the CSL's artifacts, and a code generator of 116 lines of code is used to generate UNIFY artifacts.

- For the **Parental Control CSL**, we implemented the domain concepts and relations of Figure 6.4 on page 160 in JAVA. This required a total of 10 classes and 149 lines of code. A parser of 109 lines of code is used to parse the CSL's artifacts, and a code generator of 214 lines of code is used to generate UNIFY artifacts.

There is currently little code reuse between the different CSLs (only 1 shared class of 15 lines of code). This can be improved through the future work we identified in Section 6.3.

This concludes our discussion of the UNIFY implementation. In the remainder of this chapter, we present a validation of our approach.

7.2 Validation

Over the years, WS-BPEL has become the de facto standard for specifying executable workflows. Therefore, we will use a WS-BPEL implementation of the order handling workflow (cf. Figure 4.1 on page 59) as the basis for our validation. In Section 7.2.1, we evaluate the expressiveness of UNIFY's basic connectors by comparing the UNIFY implementation of two concerns (which are to be performed *after* and *around* a number of activities, respectively) with their corresponding implementations in standard WS-BPEL and in AO4BPEL. In Section 7.2.2, we evaluate the expressiveness of UNIFY's more advanced connectors by comparing the UNIFY implementation of two more concerns (which are to be performed in *parallel* with a number of activities and by *synchronizing* two branches of a parallel control structure, respectively) with their corresponding implementations in standard WS-BPEL and in AO4BPEL. Finally, in Section 7.2.3, we evaluate the performance and scalability of UNIFY by measuring the weaving and execution times of a number of example compositions of increasing complexity, and comparing the execution times with the corresponding standard WS-BPEL processes' execution times.

7.2.1 Expressiveness of UNIFY: Basic Connectors

In this section, we show the implementation in UNIFY of two basic connectors, i.e., connectors that correspond to accepted advice types in aspect-oriented workflow languages, and compare this implementation with other approaches. For these other approaches, we choose standard WS-BPEL as representative of traditional workflow languages, and AO4BPEL as representative of existing aspect-oriented workflow languages.

In our discussion, we will focus both on quantitative (lines of code) and qualitative aspects of the compared artifacts. With regard to the quantitative aspects, we expect an improvement when adopting an aspect-oriented approach such as AO4BPEL or UNIFY instead of standard WS-BPEL. Quantitative differences between AO4BPEL and UNIFY are expected to be small. UNIFY is intended to offer the same quantitative advantages as AO4BPEL, but to differentiate itself from AO4BPEL by improving on it with regard to qualitative aspects such as separation of concerns and reusability of artifacts.

7.2.1.1 The Reporting Concern

As a first example, let us consider the order handling workflow's *reporting* concern. As is shown in the BPMN diagram in Figure 4.1, a `Report` activity has to be executed after the user has confirmed his/her order, after any of the payment activities has been executed, or after any of the shipment activities has been performed. Each of these `Report` activities is actually a composite activity, which consists of several atomic activities. In our WS-BPEL implementation of the order handling workflow, each `Report` activity is implemented as a WS-BPEL `<scope>` activity that defines a variable that will contain some input to a reporting service, an `<assign>` activity that copies context information to this variable, and an `<invoke>` activity that invokes the reporting service with the variable as its input. Listing 7.1 shows the `Report` activity for the order confirmation. The activity

consists of 39 lines of WS-BPEL code.[3] The other `Report` activities are similar to the one for the order confirmation; the main difference is the context information that is copied to the variable. The *reporting* concern is thus obviously crosscutting, with code duplication as a result. In total, the *reporting* concern is implemented in standard WS-BPEL using $6 * 39 = 234$ lines of code.

```
<scope name="ReportOrderConfirmation">
  <partnerLinks>
    <partnerLink name="ReportingBackEndPartnerLink"
      partnerLinkType="rbe:ReportingBackEndPartnerLinkType" partnerRole="me" />
  </partnerLinks>
  <variables>
    <variable name="backEndReportInput" messageType="rbe:reportRequest" />
    <variable name="backEndReportOutput" messageType="rbe:reportResponse" />
  </variables>
  <sequence>
    <assign>
      <copy>
        <from>
          <literal>
            <report xmlns="http://back_end.reporting.examples
              .unify_framework.org/xsd">
              <message>Confirm has been executed</message>
              <username />
            </report>
          </literal>
        </from>
        <to variable="backEndReportInput" part="parameters" />
      </copy>
      <copy>
        <from variable="user">
          <query xmlns:bed="http://back_end.order_books.examples
            .unify_framework.org/xsd">//bed:username</query>
        </from>
        <to variable="backEndReportInput" part="parameters">
          <query xmlns:rbed="http://back_end.reporting.examples
            .unify_framework.org/xsd">//rbed:report/rbed:username</query>
        </to>
      </copy>
    </assign>
    <invoke name="Report" partnerLink="ReportingBackEndPartnerLink"
      portType="rbe:ReportingBackEndPortType" operation="report"
      inputVariable="backEndReportInput" outputVariable="backEndReportOutput" />
  </sequence>
</scope>
```

Listing 7.1: `Report` activity for order confirmation as implemented in WS-BPEL

Existing aspect-oriented approaches for workflows address crosscutting concerns by enabling the definition of *aspects* that modularize previously crosscutting behavior. This behavior (called *advice*) can be inserted at certain locations (called *joinpoints*) in the base workflow, by selecting these joinpoints using a *pointcut*. Thus, AO4BPEL can be used to define an aspect that contains the reporting code. Because the exact same code is then being used at every joinpoint, we need a means of accessing each joinpoint's context information. This is accomplished using AO4BPEL's `ThisJPActivity` variable, which is available for use within an aspect's advice, and will contain the joinpoint activity's context information at runtime.

[3] Due to the use of XML, most of the examples in this chapter will be rather verbose.

Listing 7.2 shows the AO4BPEL aspect that implements the *reporting* concern. Lines 2–9 define a number of partner links and variables that will be added to the base workflow. Lines 11–15 define a pointcut, and lines 16–57 define the corresponding advice. Because the advice is no longer duplicated throughout the base workflow, the *reporting* concern is now implemented using a total of 59 lines of code. However, note that the concern is now implemented using a different kind of module, i.e., an aspect, than the kind of module offered by the base language, i.e., the activity. Within an aspect, new language elements introduced by AO4BPEL (cf. the `<aspect>` element on lines 1 and 59, the `<pointcutandadvice>` element on lines 10 and 58, the `<pointcut>` element on lines 11–15, and the `<advice>` element on lines 16 and 57) are intertwined with existing BPEL language elements (cf. the `<partnerLinks>` and `<variables>` elements on lines 2–9, and the advice body on lines 17–56). Even in the advice body, which consists of standard BPEL code, the use of the AO4BPEL-specific `ThisJPActivity` variable (cf. line 32) constitutes a strong dependency on the AO4BPEL approach.

```
1  <aspect name="ReportingAspect">
2    <partnerLinks>
3      <partnerLink name="ReportingBackEndPartnerLink"
4        partnerLinkType="rbe:ReportingBackEndPartnerLinkType" partnerRole="me" />
5    </partnerLinks>
6    <variables>
7      <variable name="backEndReportInput" messageType="rbe:reportRequest" />
8      <variable name="backEndReportOutput" messageType="rbe:reportResponse" />
9    </variables>
10   <pointcutandadvice>
11     <pointcut name="ActivitiesToBeReported">
12       //invoke[@name="Confirm" or @name="CreditCardPayment"
13         or @name="PayPalPayment" or @name="WireTransferPayment"
14         or @name="ShipByMail" or @name="ShipByCourier"]
15     </pointcut>
16     <advice type="after">
17       <sequence>
18         <assign>
19           <copy>
20             <from>
21               <literal>
22                 <report xmlns="http://back_end.reporting.examples
23                   .unify_framework.org/xsd">
24                   <message />
25                   <username />
26                 </report>
27               </literal>
28             </from>
29             <to variable="backEndReportInput" part="parameters" />
30           </copy>
31           <copy>
32             <from>concat($ThisJPActivity.name,
33               ' has been executed')</from>
34             <to variable="backEndReportInput" part="parameters">
35               <query xmlns:rbed="http://back_end.reporting.examples
36                 .unify_framework.org/xsd">
37                 //rbed:report/rbed:message</query>
38             </to>
39           </copy>
40           <copy>
41             <from variable="user">
42               <query xmlns:bed="http://back_end.order_books.examples
43                 .unify_framework.org/xsd">//bed:username</query>
44             </from>
45             <to variable="backEndReportInput" part="parameters">
46               <query xmlns:rbed="http://back_end.reporting.examples
```

```
47                .unify_framework.org/xsd">
48                //rbed:report/rbed:username</query>
49              </to>
50            </copy>
51          </assign>
52          <invoke name="Report" partnerLink="ReportingBackEndPartnerLink"
53            portType="rbe:ReportingBackEndPortType" operation="report"
54            inputVariable="backEndReportInput"
55            outputVariable="backEndReportOutput" />
56        </sequence>
57      </advice>
58    </pointcutandadvice>
59  </aspect>
```

Listing 7.2: Reporting aspect as implemented in AO4BPEL

All of this has an impact on possibilities for reuse of aspects in AO4BPEL. Assume that we want to invoke the reporting concern *explicitly* in another base workflow (i.e., we do not want to implicitly apply the concern using an aspect). This is impossible in AO4BPEL due to the introduction of a separate aspect module, and UNIFY will address this limitation by removing the distinction between the modules of the base language and those of the aspect mechanism. Assume that we want to apply the aspect implementing the reporting concern to another base workflow. This is impossible in AO4BPEL due to the pointcut and advice being tightly coupled to each other by being specified in the same aspect, and due to the lack of support for aspect inheritance or invoking an existing advice from within another aspect. UNIFY will address this limitation by making a clear distinction between a concern's behavior (implemented as a CompositeActivity) and a concern's deployment within a concrete context (implemented as a Connector).[4]

UNIFY differentiates itself from existing aspect-oriented approaches for workflows by offering a uniform modularization mechanism, in which crosscutting code is specified using the existing language constructs of the base language. Thus, the *reporting* concern is not implemented using a separate aspect construct, but using the existing WS-BPEL constructs for specifying composite activities, i.e., separate WS-BPEL processes or WS-BPEL `<scope>` activities. For example, Listing 7.3 shows the reporting concern as implemented in WS-BPEL for use by UNIFY.[5] Thus, specifying code that will be used in an aspect-oriented way is no different than specifying other code, except that the code is no longer crosscutting. The actual connection between the base workflow and the other modules that are to be applied to it, is specified in separate *connectors*. For example, the connector in Listing 7.4 specifies that the `Report` activity of Listing 7.3 has to be inserted *after* each joinpoint of a certain pointcut. In UNIFY, context information of the base workflow is provided to the connected concern using a *data mapping*. Because the actual workflow behavior and the connection logic are now cleanly separated, the workflow behavior can be reused, both by regular workflows or by other connectors. When comparing the lines of code of the *reporting* concern as implemented using UNIFY, i.e., 43 + 8 = 51, we obtain a significant improvement compared to standard WS-

[4] In PADUS, we addressed this limitation in another way, i.e., by supporting a rudimentary form of reuse of pointcuts and advices through `<include>` declarations (cf. Section 3.3.3).

[5] This code is the same as the WS-BPEL code from Listing 7.1, except that the message and username which will be reported are now assigned from the `reportInput` variable, which is not defined within the scope and will thus need to be mapped to an existing variable by the connector in Listing 7.4.

BPEL. We need about the same amount of code than AO4BPEL, but also get a reusable implementation: the previously crosscutting behavior is now nicely modularized in a separate CompositeActivity (the WS-BPEL <scope> activity of Listing 7.3), while its deployment is modularized in a separate Connector (the *after* connector of Listing 7.4). The same concern can be reused in different contexts by simply writing a new Connector that applies the existing CompositeActivity to another base workflow. Table 7.1 provides an overview of the lines of code in each implementation.

```
<scope name="Report">
  <partnerLinks>
    <partnerLink name="ReportingBackEndPartnerLink"
      partnerLinkType="rbe:ReportingBackEndPartnerLinkType" partnerRole="me" />
  </partnerLinks>
  <variables>
    <variable name="backEndReportInput" messageType="rbe:reportRequest" />
    <variable name="backEndReportOutput" messageType="rbe:reportResponse" />
  </variables>
  <sequence>
    <assign>
      <copy>
        <from>
          <literal>
            <report xmlns="http://back_end.reporting.examples
              .unify_framework.org/xsd">
              <message />
              <username />
            </report>
          </literal>
        </from>
        <to variable="backEndReportInput" part="parameters" />
      </copy>
      <copy>
        <from variable="reportInput" part="message" />
        <to variable="backEndReportInput" part="parameters">
          <query xmlns:rbed="http://back_end.reporting.examples
            .unify_framework.org/xsd">//rbed:report/rbed:message</query>
        </to>
      </copy>
      <copy>
        <from variable="reportInput" part="username" />
        <to variable="backEndReportInput" part="parameters">
          <query xmlns:rbed="http://back_end.reporting.examples
            .unify_framework.org/xsd">//rbed:report/rbed:username</query>
        </to>
      </copy>
    </assign>
    <invoke name="Report" partnerLink="ReportingBackEndPartnerLink"
      portType="rbe:ReportingBackEndPortType" operation="report"
      inputVariable="backEndReportInput" outputVariable="backEndReportOutput" />
  </sequence>
</scope>
```

Listing 7.3: Reporting concern as implemented in WS-BPEL for use by UNIFY

7.2.1.2 The Access Control Concern

As a second example, let us consider the order handling workflow's *access control* concern, which we used as an example for our Access Control CSL in Section 6.4. The access control concern specifies that only premium customers are allowed to execute the `SpecifyOptions` activity, and that this activity has to be skipped for other customers.

```
1 ReportConnector:
2 CONNECT Report
3 AFTER activity("SelectBooks\.Confirm|Pay\..*Payment|Ship\.ShipBy.*")
4 WITH messageTypeVariable(reportInput, message)
5          = "{$thisJoinPoint.activityName} has been executed",
6      messageTypeVariable(reportInput, username)
7          = typeVariable(user, "//bed:username", "bed",
8                         "http://back_end.order_books.examples.unify_framework.org/xsd")
```

Listing 7.4: UNIFY connector for the reporting concern

	WS-BPEL	AO4BPEL	UNIFY	UNIFY CSLs
Reporting	234	59	51	n/a
Access control	61	68	67	17
Preference saving	106	73	66	n/a
Bank account verification	58	n/a	45	n/a

Table 7.1: Comparison of lines of code required to implement the example concerns in WS-BPEL, AO4BPEL, UNIFY, and UNIFY CSLs

The access control concern also assigned these "premium customer" and "normal customer" roles to users. This concern can be implemented in WS-BPEL as listed in Listing 7.5 by inserting a `<scope>` activity around the `SpecifyOptions` activity. The `<scope>` activity will define and assign a variable containing a mapping from activities and usernames to permissions, and will subsequently use this mapping to decide whether the `SpecifyOptions` activity has to be executed or not. This is accomplished using 61 lines of WS-BPEL code.[6]

```
1  <scope name="AccessControlledSpecifyOptions">
2    <variables>
3      <variable element="nd:nestedDictionary" name="PermissionsDb"
4        xmlns:nd="http://unify-framework.org/Util/NestedDictionary"/>
5      <variable name="Action" type="xsd:string"/>
6      <variable name="Username" type="xsd:string"/>
7    </variables>
8    <sequence>
9      <assign name="InitializePermissionsDatabase">
10       <copy>
11         <from>
12           <literal>
13             <nestedDictionary
14               xmlns="http://unify-framework.org/Util/NestedDictionary">
15               <firstKey key="SpecifyOptions">
16                 <secondKey key="john">
17                   <value value="Allow"/>
18                 </secondKey>
19                 <secondKey key="mike">
20                   <value value="DenyBySkipping"/>
21                 </secondKey>
22               </firstKey>
23             </nestedDictionary>
24           </literal>
25         </from>
```

[6] Listing 7.5 lists 62 lines of code, of which 1 is a placeholder for the `SpecifyOptions` activity, so $62-1=61$ lines of code define the access control concern proper.

```
26          <to variable="PermissionsDb"/>
27        </copy>
28      </assign>
29      <assign name="InitializeUsername">
30        <copy>
31          <from>$user/bed:username</from>
32          <to variable="Username"/>
33        </copy>
34      </assign>
35      <assign name="VerifyPermissions">
36        <copy>
37          <from xmlns:nd="http://unify-framework.org/Util/NestedDictionary">
38            $PermissionsDb/nd:firstKey[@key='SpecifyOptions']
39              /nd:secondKey[@key=$Username]/nd:value/@value
40          </from>
41          <to variable="Action"/>
42        </copy>
43      </assign>
44      <if name="AC">
45        <condition>$Action='DenyBySkipping'</condition>
46        <sequence>
47          <empty name="DoNothing"/>
48        </sequence>
49        <elseif>
50          <condition>$Action='DenyByRaisingError'</condition>
51          <sequence>
52            <throw faultName="PermissionDenied" name="RaiseError"/>
53          </sequence>
54        </elseif>
55        <else>
56          <sequence>
57            <!-- ACTIVITY TO WHICH ACCESS SHOULD BE CONTROLLED (i.e., SpecifyOptions) -->
58          </sequence>
59        </else>
60      </if>
61    </sequence>
62  </scope>
```

Listing 7.5: Access control concern as implemented in WS-BPEL

The same can be accomplished using AO4BPEL as listed in Listing 7.6. In this case, the access control behavior is implemented as the body of the advice, which is inserted around the `SpecifyOptions` joinpoint. Within the body of the advice, the `<proceed>` activity that is offered by AO4BPEL can be used to specify where the joinpoint's original behavior has to be executed. This is all accomplished using 68 lines of AO4BPEL code. Using this approach, the access control concern is nicely modularized in a separate aspect. However, the concern is again implemented using a different kind of module — an aspect — which may hamper reuse as discussed above.

```
1  <aspect name="AccessControlAspect">
2    <variables>
3      <variable element="nd:nestedDictionary" name="PermissionsDb"
4        xmlns:nd="http://unify-framework.org/Util/NestedDictionary"/>
5      <variable name="Action" type="xsd:string"/>
6      <variable name="Username" type="xsd:string"/>
7    </variables>
8    <pointcutandadvice>
9      <pointcut name="ActivitiesToWhichAccessShouldBeControlled">
10       //scope[@name="SpecifyOptions"]
11     </pointcut>
12     <advice type="around">
13       <sequence>
14         <assign name="InitializePermissionsDatabase">
```

```
15          <copy>
16            <from>
17              <literal>
18                <nestedDictionary
19                  xmlns="http://unify-framework.org/Util/NestedDictionary">
20                  <firstKey key="SpecifyOptions">
21                    <secondKey key="john">
22                      <value value="Allow"/>
23                    </secondKey>
24                    <secondKey key="mike">
25                      <value value="DenyBySkipping"/>
26                    </secondKey>
27                  </firstKey>
28                </nestedDictionary>
29              </literal>
30            </from>
31            <to variable="PermissionsDb"/>
32          </copy>
33        </assign>
34        <assign name="InitializeUsername">
35          <copy>
36            <from>$user/bed:username</from>
37            <to variable="Username"/>
38          </copy>
39        </assign>
40        <assign name="VerifyPermissions">
41          <copy>
42            <from xmlns:nd="http://unify-framework.org/Util/NestedDictionary">
43              $PermissionsDb/nd:firstKey[@key=$ThisJPActivity.name]
44                /nd:secondKey[@key=$Username]/nd:value/@value</from>
45            <to variable="Action"/>
46          </copy>
47        </assign>
48        <if name="AC">
49          <condition>$Action='DenyBySkipping'</condition>
50          <sequence>
51            <empty name="DoNothing"/>
52          </sequence>
53          <elseif>
54            <condition>$Action='DenyByRaisingError'</condition>
55            <sequence>
56              <throw faultName="PermissionDenied" name="RaiseError"/>
57            </sequence>
58          </elseif>
59          <else>
60            <sequence>
61              <proceed />
62            </sequence>
63          </else>
64        </if>
65      </sequence>
66    </advice>
67  </pointcutandadvice>
68 </aspect>
```

Listing 7.6: Access control concern as implemented in AO4BPEL

UNIFY can be used to implement the access control concern by specifying a composite activity and a connector, with the advantages we already mentioned while discussing the reporting concern above. However, we can also use our Access Control CSL to implement this concern, as was shown in Listing 6.1 in Section 6.4. This requires only 17 lines of XML code, which is more closely related to the domain concepts of the concern than the corresponding WS-BPEL, AO4BPEL or UNIFY implementations. This UNIFY implementation can be generated by the Access Control CSL preprocessor. For the access

control concern's behavior, a WS-BPEL <scope> activity is generated that consists of 61 lines of WS-BPEL code (cf. Listing 7.7). Note that this is all standard WS-BPEL code: unlike AO4BPEL, the behavior does not contain custom activities such as <proceed>. For the connection logic, a UNIFY around connector is generated that consists of 6 lines of code (i.e., the connector that was already provided in Listing 6.2, augmented with 2 lines to map data; cf. Listing 7.8). When comparing the lines of code of the *access control* concern as implemented using UNIFY, i.e., 61 + 6 = 67, with the standard WS-BPEL implementation, we do not achieve an improvement because the access control concern was only applied to one activity, i.e., SpecifyOptions. The same holds for the AO4BPEL implementation. However, both AO4BPEL and UNIFY offer improved separation of concerns. With UNIFY, we need about the same amount of code than AO4BPEL, but also get a reusable implementation. The Access Control CSL implementation, however, requires less code than all of these. Table 7.1 provides an overview of the lines of code in each implementation.

```
<scope name="GeneratedAccessControlActivity">
  <variables>
    <variable element="nd:nestedDictionary" name="PermissionsDb"
      xmlns:nd="http://unify-framework.org/Util/NestedDictionary"/>
    <variable name="Action" type="xsd:string"/>
    <variable name="Username" type="xsd:string"/>
  </variables>
  <sequence>
    <assign name="InitializePermissionsDatabase">
      <copy>
        <from>
          <literal>
            <nestedDictionary
              xmlns="http://unify-framework.org/Util/NestedDictionary">
              <firstKey key="SpecifyOptions">
                <secondKey key="john">
                  <value value="Allow"/>
                </secondKey>
                <secondKey key="mike">
                  <value value="DenyBySkipping"/>
                </secondKey>
              </firstKey>
            </nestedDictionary>
          </literal>
        </from>
        <to variable="PermissionsDb"/>
      </copy>
    </assign>
    <assign name="InitializeUsername">
      <copy>
        <from>$user/bed:username</from>
        <to variable="Username"/>
      </copy>
    </assign>
    <assign name="VerifyPermissions">
      <copy>
        <from xmlns:nd="http://unify-framework.org/Util/NestedDictionary">
          $PermissionsDb/nd:firstKey[@key=$Activity]
            /nd:secondKey[@key=$Username]/nd:value/@value</from>
        <to variable="Action"/>
      </copy>
    </assign>
    <if name="AC">
      <condition>$Action='DenyBySkipping'</condition>
      <sequence>
        <empty name="DoNothing"/>
```

```
47        </sequence>
48        <elseif>
49          <condition>$Action='DenyByRaisingError'</condition>
50          <sequence>
51            <throw faultName="PermissionDenied" name="RaiseError"/>
52          </sequence>
53        </elseif>
54        <else>
55          <sequence>
56            <empty name="GeneratedDummyActivity" />
57          </sequence>
58        </else>
59      </if>
60    </sequence>
61  </scope>
```

Listing 7.7: Access control concern as implemented in WS-BPEL for use by UNIFY

```
1 GeneratedAccessControlConnector:
2 CONNECT GeneratedAccessControlActivity
3 AROUND activity("OrderHandling\.SpecifyOptions")
4 PROCEEDING AT activity("GeneratedAccessControlActivity\.GeneratedDummyActivity")
5 WITH typeVariable(Activity)
6          = messageTypeVariable(thisJoinPoint, activityName)
```

Listing 7.8: UNIFY connector for the access control concern

7.2.2 Expressiveness of UNIFY: Advanced Connectors

In this section, we show the implementation in UNIFY of two advanced connectors, i.e., connectors that constitute novel advice types with regard to existing aspect-oriented workflow languages, and compare this implementation with other approaches. Once again, we choose standard WS-BPEL as representative of traditional workflow languages, and AO4BPEL as representative of existing aspect-oriented workflow languages.

7.2.2.1 The Preference Saving Concern

As a first example of an advanced connector, we consider the order handling workflow's *preference saving* concern. As is shown in the BPMN diagram in Figure 4.1, a *SavePreference* activity has to be executed in parallel with searching for a book and adding a book to the shopping basket. Each of these two *SavePreference* activities is actually a composite activity, which consists of several atomic activities. In our WS-BPEL implementation of the order handling workflow, a *SavePreference* activity is implemented as a WS-BPEL `<scope>` activity that defines a variable that will contain some input to a preferences service, an `<assign>` activity that copies context information to this variable, and an `<invoke>` activity that invokes the preferences service with the variable as its input. The activity consists of 53 lines of WS-BPEL code (cf. Listing 7.9). The *SavePreference* activity for searching a book is similar to the one for adding a book; the main difference is the context information that is copied to the variable. The *preference saving* concern is thus crosscutting, with code duplication as a result. In total, the *preference saving* concern is implemented in standard WS-BPEL using $2 * 53 = 106$ lines of code.

```
<scope name="SavePreferenceForAddedBook"
  <partnerLinks>
    <partnerLink name="PreferencesBackEndPartnerLink"
      partnerLinkType="pbe:PreferencesBackEndPartnerLinkType" partnerRole="me" />
  </partnerLinks>
  <variables>
    <variable name="backEndSaveUserOperationInput"
      messageType="pbe:saveUserOperationRequest" />
    <variable name="backEndSaveUserOperationOutput"
      messageType="pbe:saveUserOperationResponse" />
  </variables>
  <sequence>
    <assign>
      <copy>
        <from>
          <literal>
            <report xmlns="http://back_end.preferences.examples
              .unify_framework.org/xsd">
              <username />
              <operation>AddBook</operation>
              <bookId />
            </report>
          </literal>
        </from>
        <to variable="backEndSaveUserOperationInput" part="parameters" />
      </copy>
      <copy>
        <from variable="user">
          <query xmlns:bed="http://back_end.order_books.examples
            .unify_framework.org/xsd">//bed:username</query>
        </from>
        <to variable="backEndSaveUserOperationInput" part="parameters">
          <query xmlns:pbed="http://back_end.preferences.examples
            .unify_framework.org/xsd">//pbed:saveUserOperation/pbed:username</query>
        </to>
      </copy>
      <copy>
        <from variable="addBookInput" part="parameters">
          <query xmlns:bed="http://back_end.order_books.examples
            .unify_framework.org/xsd">//bed:addBook/bed:id</query>
        </from>
        <to variable="backEndSaveUserOperationInput" part="parameters">
          <query xmlns:pbed="http://back_end.preferences.examples
            .unify_framework.org/xsd">//pbed:saveUserOperation/pbed:bookId</query>
        </to>
      </copy>
    </assign>
    <invoke name="SaveUserOperation" partnerLink="PreferencesBackEndPartnerLink"
      portType="pbe:PreferencesBackEndPortType" operation="saveUserOperation"
      inputVariable="backEndSaveUserOperationInput"
      outputVariable="backEndSaveUserOperationOutput" />
  </sequence>
</scope>
```

Listing 7.9: Preference saving concern as implemented in WS-BPEL

The same can be accomplished using AO4BPEL. In this case, the preference saving behavior is implemented as the body of the advice, which is inserted in parallel with the *SavePreference* joinpoint using an around advice that defines a new parallel control structure. This is all accomplished using 73 lines of AO4BPEL code (cf. Listing 7.10). Using this approach, the preference saving concern is nicely modularized in a separate aspect. However, this aspect contains both the behavior that is to be inserted, and the connection logic that specifies where the behavior has to be inserted. For example, this

aspect cannot be reused to perform preference saving *before* or *after* a joinpoint.

```
<aspect name="PreferenceSavingAspect">
  <partnerLinks>
    <partnerLink name="PreferencesBackEndPartnerLink"
      partnerLinkType="pbe:PreferencesBackEndPartnerLinkType" partnerRole="me" />
  </partnerLinks>
  <variables>
    <variable name="backEndSaveUserOperationInput"
      messageType="pbe:saveUserOperationRequest" />
    <variable name="backEndSaveUserOperationOutput"
      messageType="pbe:saveUserOperationResponse" />
  </variables>
  <pointcutandadvice>
    <pointcut name="ActivitiesForWhichPreferencesMustBeSaved">
      //invoke[@name="SearchBook" or @name="AddBook"]
    </pointcut>
    <advice type="around">
      <flow>
        <proceed />
        <sequence>
          <assign>
            <copy>
              <from>
                <literal>
                  <report xmlns="http://back_end.preferences.examples
                    .unify_framework.org/xsd">
                    <username />
                    <operation />
                    <bookId />
                  </report>
                </literal>
              </from>
              <to variable="backEndSaveUserOperationInput" part="parameters" />
            </copy>
            <copy>
              <from variable="user">
                <query xmlns:bed="http://back_end.order_books.examples
                  .unify_framework.org/xsd">//bed:username</query>
              </from>
              <to variable="backEndSaveUserOperationInput" part="parameters">
                <query xmlns:pbed="http://back_end.preferences.examples
                  .unify_framework.org/xsd">
                  //pbed:saveUserOperation/pbed:username</query>
              </to>
            </copy>
            <copy>
              <from variable="ThisJPActivity" part="name" />
              <to variable="backEndSaveUserOperationInput" part="parameters">
                <query xmlns:pbed="http://back_end.preferences.examples
                  .unify_framework.org/xsd">
                  //pbed:saveUserOperation/pbed:operation</query>
              </to>
            </copy>
            <copy>
              <from variable="addBookInput" part="parameters">
                <query xmlns:bed="http://back_end.order_books.examples
                  .unify_framework.org/xsd">//bed:addBook/bed:id</query>
              </from>
              <to variable="backEndSaveUserOperationInput" part="parameters">
                <query xmlns:pbed="http://back_end.preferences.examples
                  .unify_framework.org/xsd">
                  //pbed:saveUserOperation/pbed:bookId</query>
              </to>
            </copy>
          </assign>
          <invoke name="SaveUserOperation" partnerLink="PreferencesBackEndPartnerLink"
```

```
66              portType="pbe:PreferencesBackEndPortType" operation="saveUserOperation"
67              inputVariable="backEndSaveUserOperationInput"
68              outputVariable="backEndSaveUserOperationOutput" />
69          </sequence>
70        </flow>
71      </advice>
72    </pointcutandadvice>
73  </aspect>
```

Listing 7.10: Preference saving concern as implemented in AO4BPEL

UNIFY can be used to implement the access control concern by specifying a composite activity and a connector. The composite activity is a WS-BPEL <scope> activity that consists of 57 lines of WS-BPEL code (cf. Listing 7.11). Once again, note that this is all standard WS-BPEL code: unlike AO4BPEL, the behavior does not contain custom activities such as <proceed>. The code also does not define the connection logic that was present in AO4BPEL's advice body. The connection logic is specified in a separate parallel connector that consists of 9 lines of code (cf. Listing 7.12). When comparing the lines of code of the *preference saving* concern as implemented using UNIFY, i.e., 57+9 = 66, we obtain a significant improvement compared to standard WS-BPEL. We need about the same amount of code than AO4BPEL, but also get a reusable implementation. Table 7.1 provides an overview of the lines of code in each implementation.

```
1  <scope name="SavePreference">
2    <partnerLinks>
3      <partnerLink name="PreferencesBackEndPartnerLink"
4        partnerLinkType="pbe:PreferencesBackEndPartnerLinkType" partnerRole="me" />
5    </partnerLinks>
6    <variables>
7      <variable name="backEndSaveUserOperationInput"
8        messageType="pbe:saveUserOperationRequest" />
9      <variable name="backEndSaveUserOperationOutput"
10       messageType="pbe:saveUserOperationResponse" />
11   </variables>
12   <sequence>
13     <assign>
14       <copy>
15         <from>
16           <literal>
17             <report xmlns="http://back_end.preferences.examples
18               .unify_framework.org/xsd">
19               <username />
20               <operation />
21               <bookId />
22             </report>
23           </literal>
24         </from>
25         <to variable="backEndSaveUserOperationInput" part="parameters" />
26       </copy>
27       <copy>
28         <from variable="saveUserOperationInput" part="username" />
29         <to variable="backEndSaveUserOperationInput" part="parameters">
30           <query xmlns:pbed="http://back_end.preferences.examples
31             .unify_framework.org/xsd">
32             //pbed:saveUserOperation/pbed:username</query>
33         </to>
34       </copy>
35       <copy>
36         <from variable="saveUserOperationInput" part="operation" />
37         <to variable="backEndSaveUserOperationInput" part="parameters">
38           <query xmlns:pbed="http://back_end.preferences.examples
39             .unify_framework.org/xsd">
```

```
40            //pbed:saveUserOperation/pbed:operation</query>
41          </to>
42        </copy>
43        <copy>
44          <from variable="saveUserOperationInput" part="bookId" />
45          <to variable="backEndSaveUserOperationInput" part="parameters">
46            <query xmlns:pbed="http://back_end.preferences.examples
47              .unify_framework.org/xsd">
48              //pbed:saveUserOperation/pbed:bookId</query>
49          </to>
50        </copy>
51      </assign>
52      <invoke name="SaveUserOperation" partnerLink="PreferencesBackEndPartnerLink"
53        portType="pbe:PreferencesBackEndPortType" operation="saveUserOperation"
54        inputVariable="backEndSaveUserOperationInput"
55        outputVariable="backEndSaveUserOperationOutput" />
56    </sequence>
57  </process>
```

Listing 7.11: Preference saving concern as implemented in WS-BPEL for use by UNIFY

```
1  SavePreferenceConnector:
2  CONNECT SavePreference
3  PARALLEL TO activity("SelectBooks\..*Book")
4  WITH messageTypeVariable(saveUserOperationInput, username)
5           = typeVariable(user, "//bed:username", "bed",
                "http://back_end.order_books.examples.unify_framework.org/xsd"),
6       messageTypeVariable(saveUserOperationInput, operation)
7           = "{$thisJoinPoint.activityName}",
8       messageTypeVariable(saveUserOperationInput, bookId)
9           = messageTypeVariable(selectBookOutput, parameters,
                "//ud:selectBookResponse/ud:return/ud:id", "ud",
                "http://user.order_books.examples.unify_framework.org/xsd")
```

Listing 7.12: UNIFY connector for the preference saving concern

7.2.2.2 The Bank Account Verification Concern

As a second example of an advanced connector, we consider the order handling workflow's *bank account verification* concern. As is shown in the BPMN diagram in Figure 4.1, the *VerifyBankAccount* activity has to be executed after the *Pay* activity and before the *Ship* activity, in a way that synchronizes the two parallel branches in which these activities are present. Once again, the *VerifyBankAccount* activity is actually a composite activity, which consist of several atomic activities. In our WS-BPEL implementation of the order handling workflow, the *VerifyBankAccount* activity is implemented as a WS-BPEL `<scope>` activity that defines a variable that will contain some input to a verification service, an `<assign>` activity that copies context information to this variable, and an `<invoke>` activity that invokes the preferences service with the variable as its input. WS-BPEL is traditionally used to create workflows in a block-structured way. In order to synchronize two branches of a parallel control structure, the block-structured constructs are not sufficient and we must thus use the WS-BPEL `<link>` construct which allows creating workflows in a graph-based way. The *VerifyBankAccount* activity is thus added as a third branch to the existing parallel control structure, with two links specifying that the *VerifyBankAccount* activity has to be started after the *Pay* activity has been

executed, and that the *Ship* activity can only be executed after the *VerifyBankAccount* activity has been executed. All of this is implemented using a total of 58 lines of WS-BPEL code (cf. Listing 7.13).[7]

```
<flow>
  <links>
    <link name="LinkA"/>
    <link name="LinkB"/>
  </links>
  <scope name="VerifyBankAccount">
    <sources>
      <source linkName="LinkB"/>
    </sources>
    <targets>
      <target linkName="LinkA"/>
    </targets>
    <partnerLinks>
      <partnerLink name="OrderBooksBackEndPartnerLink"
        partnerLinkType="be:OrderBooksBackEndPartnerLinkType" partnerRole="me"/>
    </partnerLinks>
    <variables>
      <variable messageType="be:verifyBankAccountRequest"
        name="backEndVerifyBankAccountInput"/>
      <variable messageType="be:verifyBankAccountResponse"
        name="backEndVerifyBankAccountOutput"/>
    </variables>
    <sequence>
      <assign>
        <copy>
          <from>
            <literal>
              <verifyBankAccount xmlns="http://back_end.order_books.examples
                .unify_framework.org/xsd">
                <username/>
              </verifyBankAccount>
            </literal>
          </from>
          <to part="parameters" variable="backEndVerifyBankAccountInput"/>
        </copy>
        <copy>
          <from variable="user">
            <query xmlns:bed="http://back_end.order_books.examples
              .unify_framework.org/xsd">//bed:username</query>
          </from>
          <to part="parameters" variable="backEndVerifyBankAccountInput">
            <query xmlns:bed="http://back_end.order_books.examples
              .unify_framework.org/xsd">
              //bed:verifyBankAccount/bed:username</query>
          </to>
        </copy>
      </assign>
      <invoke name="VerifyBankAccount" partnerLink="OrderBooksBackEndPartnerLink"
        portType="be:OrderBooksBackEndPortType" operation="verifyBankAccount"
        inputVariable="backEndVerifyBankAccountInput"
        outputVariable="backEndVerifyBankAccountOutput" />
    </sequence>
  </scope>
  <sequence>
    <empty name="Pay">
      <sources>
```

[7] Listing 7.13 lists 70 lines of code, of which 2 define the `<flow>` activity surrounding the bank account verification concern, 5 define the sequence containing placeholders for the `Pay` and `SendInvoice` activities, and 5 define the sequence containing placeholders for the `ProcessOrder` and `Ship` activities, so $70-2-5-5=58$ lines of code define the bank account verification concern proper.

```
57          <source linkName="LinkA"/>
58        </sources>
59      </empty>
60      <empty name="SendInvoice" />
61    </sequence>
62    <sequence>
63      <empty name="ProcessOrder" />
64      <empty name="Ship">
65        <targets>
66          <target linkName="LinkB"/>
67        </targets>
68      </empty>
69    </sequence>
70  </flow>
```

Listing 7.13: Bank account verification concern as implemented in WS-BPEL

This behavior cannot be modularized in a separate aspect using AO4BPEL. UNIFY, however, does support this using its *synchronizing* connector. Once again, the actual behavior is implemented as a composite activity, which is a WS-BPEL <scope> activity that consists of 38 lines of WS-BPEL code (cf. Listing 7.14). The connection logic is specified in a separate synchronizing connector that consists of 7 lines of code (cf. Listing 7.15). When comparing the lines of code of the *bank account verification* concern as implemented using UNIFY, i.e., 38 + 7 = 45, we achieve an improvement over standard WS-BPEL, because the way in which WS-BPEL <link> constructs are specified is rather verbose, whereas our connector construct abstracts over this. Additionally, UNIFY offers full separation of concerns because the concern is now nicely modularized.

```
1  <scope name="VerifyBankAccount">
2    <partnerLinks>
3      <partnerLink name="OrderBooksBackEndPartnerLink"
4        partnerLinkType="be:OrderBooksBackEndPartnerLinkType" partnerRole="me" />
5    </partnerLinks>
6    <variables>
7      <variable name="backEndVerifyBankAccountInput"
8        messageType="be:verifyBankAccountRequest" />
9      <variable name="backEndVerifyBankAccountOutput"
10       messageType="be:verifyBankAccountResponse" />
11   </variables>
12   <sequence>
13     <assign>
14       <copy>
15         <from>
16           <literal>
17             <verifyBankAccount xmlns="http://back_end.order_books.examples
18               .unify_framework.org/xsd">
19               <username />
20             </verifyBankAccount>
21           </literal>
22         </from>
23         <to variable="backEndVerifyBankAccountInput" part="parameters" />
24       </copy>
25       <copy>
26         <from variable="verifyBankAccountInput" part="username" />
27         <to variable="backEndVerifyBankAccountInput" part="parameters">
28           <query xmlns:bed="http://back_end.order_books.examples
29             .unify_framework.org/xsd">//bed:verifyBankAccount/bed:username</query>
30         </to>
31       </copy>
32     </assign>
33     <invoke name="VerifyBankAccount" partnerLink="OrderBooksBackEndPartnerLink"
34       portType="be:OrderBooksBackEndPortType" operation="verifyBankAccount"
```

```
35        inputVariable="backEndVerifyBankAccountInput"
36        outputVariable="backEndVerifyBankAccountOutput" />
37    </sequence>
38  </scope>
```

Listing 7.14: Bank account verification concern as implemented in WS-BPEL for use by UNIFY

```
1  VerifyBankAccountConnector:
2  CONNECT VerifyBankAccount
3  IN fragment(OrderHandling.PaymentAndShipmentSplit, OrderHandling.PaymentAndShipmentJoin)
4  AND-SPLITTING AT activity("OrderHandling\.Pay")
5  SYNCHRONIZING AT activity("OrderHandling\.Ship")
6  WITH messageTypeVariable(verifyBankAccountInput, username)
7           = typeVariable(user, "//bed:username", "bed",
                 "http://back_end.order_books.examples.unify_framework.org/xsd")
```

Listing 7.15: UNIFY connector for the bank account verification concern

7.2.3 Performance and Scalability of UNIFY

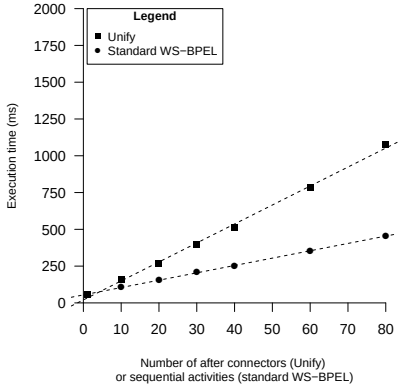
In this section, we measure the performance and scalability of the UNIFY framework. We compare the execution time of two UNIFY workflow compositions — one containing after connectors, and one containing parallel connectors — with their equivalent implementations in standard WS-BPEL for increasing amounts of connectors. Because AO4BPEL uses a modified WS-BPEL engine, we do not include it in this comparison. We also measure the weaving time of the UNIFY workflow compositions.

Our experiments are performed on an Apple MacBook Pro with a 2.66 GHz Intel Core i7-620M processor and 8 GB of 1066 MHz DDR3 RAM. The operating system is Max OS X (version 10.7.3), the installed Java VM is version 1.6.0 update 33, and the workflows are executed using the Apache ODE WS-BPEL engine (version 1.3.5), deployed as a web application on Apache Tomcat (version 7.0.28). The execution time is measured by a separate Java application that invokes the workflows that are deployed on ODE. Each workflow's execution is measured 10 times, the median of which is included in our final results. Before measuring is commenced, all workflows are invoked 10 times in order to "warm-up" the Java VM. Results are analyzed using the R environment for statistical computing.[8]

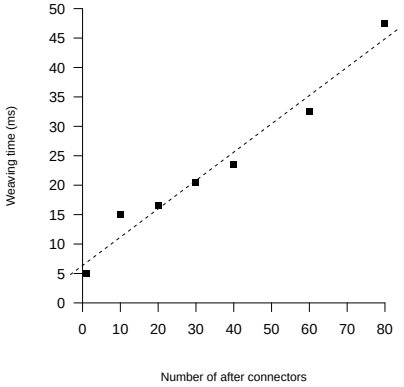
7.2.3.1 Scalability of Basic Connectors

In order to gauge the scalability of UNIFY's basic connectors, we perform several experiments in which an increasing number of *after* connectors (i.e., 1, 10, 20, 30, 40, 60, and 80 connectors) are used to insert an advice activity after the same activity in a base workflow. We then compare the execution time of the woven composition with a standard WS-BPEL workflow in which all these activities have been combined manually. The activities are all invocations of a single service that immediately returns a response. Thus,

[8] Cf. http://www.r-project.org/



(a) Comparing the execution time of workflows created using an increasing number of UNIFY *after* connectors with the equivalent manually implemented WS-BPEL workflows



(b) Weaving time for compositions with an increasing number of UNIFY *after* connectors

Figure 7.2: Measurement of runtime and weaving overhead introduced by UNIFY *after* connectors

the amount of actual work to be performed by the workflow is the same in both the UNIFY implementation and the standard WS-BPEL implementation, and differences in measured time can thus be attributed to the time spent on the *control flow* of the experiments' workflows.

Figure 7.2(a) plots the measurements for our experiments regarding *after* connectors. For both UNIFY and standard WS-BPEL, there exists a linear correlation between the workflow execution time and the number of connectors or sequential activities. The corresponding linear models are plotted in Figure 7.2(a) as dashed lines, and Table 7.2 lists the models' values for R^2, a, and b, where R^2 is the squared correlation coefficient and the linear model is defined as $ax+b$. The linear correlation between the number of connectors/activities and the execution time shows that the weaving of our *after* connectors is scalable, although UNIFY introduces an overhead (of a factor 2.58) in comparison to standard WS-BPEL, which can be attributed to the fact that the advice code inserted by a connector is always enclosed in a new WS-BPEL `<scope>` activity, whereas this additional structuring of the workflow is not present in the manually specified standard WS-BPEL workflows.

In addition to the runtime overhead introduced by UNIFY, we must also consider the overhead introduced by the weaving process. Therefore, we also measured the time required to weave the compositions used in the above experiments. These measurements are plotted in Figure 7.2(b). Once again, UNIFY is scalable, with R^2 being 0.9669, and the linear model being $0.48137x+6.35578$. The weaving time overhead is very small (about

	UNIFY	Standard WS-BPEL
R^2	0.9974	0.9995
a	12.9011	4.99695
b	20.2612	54.39080

Table 7.2: Correlation results for our experiments regarding *after* connectors (cf. Figure 7.2(a))

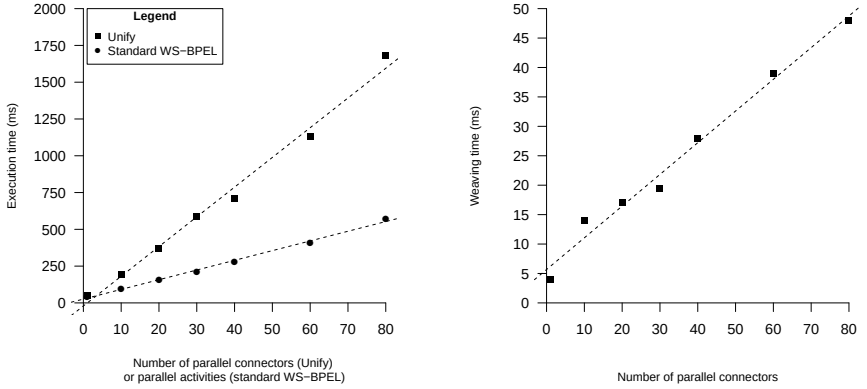
half a millisecond per connector).

7.2.3.2 Scalability of Advanced Connectors

In order to gauge the scalability of UNIFY's advanced connectors, we perform several experiments in which an increasing number of *parallel* connectors (i.e., 1, 10, 20, 30, 40, 60, and 80 connectors) are used to insert an advice activity in parallel with the same activity in a base workflow. We then compare the execution time of the woven composition with a standard WS-BPEL workflow in which all these activities have been specified manually. The activities are all invocations of a single service that immediately returns a response. Thus, the amount of actual work to be performed by the workflow is the same in both the UNIFY implementation and the standard WS-BPEL implementation, and differences in measured time can thus be attributed to the time spent on the control flow of the experiments' workflows.

Figure 7.3(a) plots the measurements for our experiments regarding *parallel* connectors. For both UNIFY and standard WS-BPEL, there exists a linear correlation between the workflow execution time and the number of connectors or parallel activities. The corresponding linear models are plotted in Figure 7.3(a) as dashed lines, and Table 7.3 lists the models' values for R^2, a, and b. The linear correlation between the number of connectors/activities and the execution time shows that the weaving of our *parallel* connectors is scalable, although UNIFY introduces an overhead (of a factor 3.07) in comparison to standard WS-BPEL, which again can be attributed to the fact that the advice code inserted by a connector is always enclosed in a new WS-BPEL <scope> activity, whereas this additional structuring of the workflow is not present in the manually specified standard WS-BPEL workflows. Additionally, the advice code inserted by a *parallel* connector is always enclosed in a new WS-BPEL `<flow>` activity, i.e., a new parallel control structure is inserted for every connector, thus obtaining a nesting of parallel control structures instead of only one control structure as in the manually specified standard WS-BPEL workflows. This latter issue, which may explain the larger overhead of *parallel* connectors compared to *after* connectors, can be improved by optimizing the weaving process to generate only one parallel control structure per joinpoint activity.

Once again, we also consider the overhead introduced by the weaving process. These measurements are plotted in Figure 7.3(b). UNIFY is scalable, with R^2 being 0.9845, and the linear model being $0.53859x + 5.67152$. The weaving time overhead is very small (about half a millisecond per connector).



(a) Comparing the execution time of workflows created using an increasing number of UNIFY *parallel* connectors with the equivalent manually implemented WS-BPEL workflows

(b) Weaving time for compositions with an increasing number of UNIFY *parallel* connectors

Figure 7.3: Measurement of runtime and weaving overhead introduced by UNIFY *parallel* connectors

	UNIFY	**Standard WS-BPEL**
R^2	0.9898	0.9963
a	20.195	6.5875
b	−21.015	25.8457

Table 7.3: Correlation results for our experiments regarding *parallel* connectors (cf. Figure 7.3(a))

7.2.4 Discussion

In the previous sections, we have provided an initial attempt at measuring the expressiveness and scalability of UNIFY. With regard to the expressiveness, we can conclude that, when implementing a given concern, UNIFY requires less code than standard WS-BPEL *if the concern is crosscutting*, i.e., the concern is to be applied at several locations in the base workflow. This is due to the fact that such concerns give rise to code duplication in WS-BPEL. If the concern is not crosscutting, UNIFY may require slightly more code because of the need to specify a separate concern and connector. In both of these cases, however, UNIFY has the advantage of better separation of concerns, as the concern is modularized in a separate module. Such decomposition is traditionally associated with managerial benefits and improved product flexibility and comprehensibility (Parnas, 1972). However, further experiments are required to verify whether these ben-

efits also apply to UNIFY. When comparing UNIFY to AO4BPEL, we can conclude that although both approaches require the same amount of code to implement a given concern, UNIFY concerns have the advantage of being more reusable, because of the strict separation between the actual concern's behavior and the connection logic. Both approaches, however, require the workflow developer to have some notion of the aspect-oriented programming paradigm. Further experiments are required to gauge whether this influences the usability of the approaches as experienced by workflow developers. When comparing our CSLs with the above approaches, we can conclude that at least the Access Control CSL requires much less code to implement a given concern, and that the abstraction level of the CSL artifact is much closer to the domain of the concern. However, further experiments are required in order to validate the usability of CSLs in our context.

With regard to the scalability, we can conclude that a composition consisting of *after* connectors scales with an increasing number of connectors. A linear runtime overhead is induced by the UNIFY weaver, because it gives rise to workflows that contain more structural information that a manually implemented workflow. The UNIFY weaver also induces a weaving time overhead, which is also linear, and is much smaller than the runtime overhead. The same observations were made with regard to our experiments involving *parallel* connectors, with the difference that the linear runtime overhead is larger than the one for *after* connectors. This difference could be mitigated by optimizing the UNIFY weaver. Although these initial experiments have a positive outcome, further experiments are required in order to verify whether compositions remain scalable when other types of connectors or combinations of different types of connectors are used, or when connectors are applied to multiple joinpoints. We also did not include CSLs in our scalability experiments, because they give rise to regular UNIFY concerns and connectors. However, the CSL preprocessors will induce an additional weaving time overhead.

Finally, all of our experiments were targeted at WS-BPEL. Further experiments should include other workflow languages such as BPMN or YAWL. Also, our current implementation is a proof-of-concept in JAVA; this implementation can be fine-tuned in order to achieve better results.

7.3 Summary

In this chapter, we first present the implementation of the UNIFY framework. At the heart of the framework lies a JAVA implementation of the base language and connector mechanism presented in Chapter 4. Instantiations towards WS-BPEL and BPMN allow importing/exporting WS-BPEL and BPMN code to/from UNIFY. The concern-specific layer built on top of UNIFY allow transforming CSL artifacts into UNIFY concerns and connectors.

Second, we perform a validation of UNIFY with respect to expressivity, performance and scalability. We discuss four example concerns and compare their implementations in standard WS-BPEL (as representative of current workflow languages), AO4BPEL (as representative of aspect-oriented workflow languages), and UNIFY (with or without CSLs). In comparison to WS-BPEL, UNIFY requires less code for implementing *crosscutting*

concerns. For both crosscutting and non-crosscutting concerns, UNIFY improves separation of concerns. In comparison to AO4BPEL, UNIFY offers improved separation of concerns and facilitates reuse of modularized code. We compare the execution time of standard WS-BPEL workflows with UNIFY workflows, and measure the weaving time of UNIFY workflows. In both cases, UNIFY scales linearly with the amount of connectors applied to the workflow.

Chapter 8

Conclusions

8.1 Summary and Contributions

In this dissertation, we focus on improving separation of concerns in workflow languages by developing an approach that allows specifying workflow concerns in isolation of each other, and subsequently composing these concerns according to a coherent collection of concern connection patterns.

We give an overview of the history of the workflow paradigm, describe the paradigm itself as well as its terminology, present business process management and web service orchestration as two important application domains, and present BPEL, BPMN and YAWL as three well-accepted workflow languages before delving into the topic of separation of concerns in workflows. We analyze the state of the art in modularization mechanisms in workflow languages, and conclude that these mechanisms fail to ensure separation of concerns, as they typically require all workflow concerns to be specified in a single, monolithic workflow specification, which precludes effective maintenance and reuse of workflow concerns. Although existing aspect-oriented approaches for workflows remedy this problem to some extent, the concern connection patterns offered by these approaches do not recognize the prevalence of elementary control flow concepts within the workflow paradigm, such as parallelism and choice. The existing approaches apply the accepted ideas of aspect-oriented programming to workflows, but do not go far beyond these ideas. We, on the other hand, propose and implement a comprehensive vision on modularization of workflow concerns by reconsidering the elementary modularization constructs of workflow languages. We thus refine our main goal into five requirements: (1) the approach must facilitate the design, evolution, and reusability of individual workflow concerns by allowing these concerns to be specified in isolation of each other, (2) the approach must allow composing these concerns according to a coherent set of workflow-specific concern connection patterns, (3) the approach must support the definition of concerns using constructs that are close to the concerns' domains, (4) the approach must be underpinned with a solid semantical foundation, and (5) the approach must be applicable to several existing workflow languages, and independent of any specific workflow engine. We iteratively fulfill these requirements during

the course of two main experiments.

Our first experiment in facilitating the effective modularization of workflow concerns has taken place in the specific context of the WIT-CASE project, which studied and validated innovative solutions for the creation, deployment and runtime execution of services on top of a novel Service Delivery Platform, which is the service infrastructure operated by a telecom service provider or network operator. Based on the characteristics of the telecom Service Delivery Platform and the goals of the WIT-CASE project, our first solution, which is named PADUS, only addresses the modularization of *crosscutting* concerns in BPEL workflows by allowing these to be specified in isolation of each other, as separate *aspects*. PADUS differentiates itself from existing aspect-oriented approaches for workflows, most notably the ones that are applicable to BPEL, by reconsidering essential language properties such as the joinpoint model, pointcut language, advice language, and aspect deployment, and by supporting an entirely different implementation strategy. In addition to the traditional aspect-oriented concern connection patterns offered by existing approaches, PADUS offers the *in* advice type that is specifically useful in a workflow context, as it allows introducing an advice activity as a parallel branch of a base workflow, as an alternative branch of a base workflow, as a fault handler or compensation handler of a base workflow, etc. All of these patterns can be applied to a rich joinpoint model that consists of all BPEL activities (i.e., not only `<invoke>` activities) using a high-level, logic pointcut language that allows selecting joinpoints in a way that alleviates, among others, the *pointcut fragility problem* associated with existing pointcut languages. Aspects are instantiated and applied to a workflow using an explicit deployment construct, which allows specifying precedence among aspects and thus prevents possible aspect composition problems. Finally, PADUS is implemented as a source code weaver which ensures full compatibility with the existing BPEL tool chain. Thus, PADUS effectively addresses the scope we identified in relation to the goals of the WIT-CASE project.

As a second experiment, we have developed a framework named UNIFY, which goes beyond the scope of PADUS and fulfills all of the requirements enumerated above. At the heart of UNIFY lies a base language meta-model that allows uniform modularization of *all* workflow concerns. Every workflow concern, be it regular or crosscutting, is independently specified, using the same language construct, i.e., the CompositeActivity. In this respect, UNIFY is related to symmetric aspect-oriented approaches such as HYPERJ (Tarr et al., 1999) and FUSEJ (Suvée et al., 2006). The base language meta-model allows expressing arbitrary workflows, and can be instantiated towards several concrete workflow languages. We introduce a coherent collection of seven external concern connection patterns (i.e., the *before*, *after*, *replace*, *around*, *parallel*, *alternative*, and *iterating* patterns) and two internal concern connection patterns (i.e., the *synchronizing parallel branches* and *switching alternative branches* patterns) that recognize the specific characteristics of workflows, including the workflow paradigm's heavy focus on parallelism and choice. We provide a connector mechanism that allows independently modularized (regular and/or crosscutting) concerns to be connected according to the above concern connection patterns. *External connectors* allow introducing behavior sequentially before, after, or around joinpoints, in parallel with joinpoints, as an alternative to joinpoints, or in iterations with joinpoints, while the workflow in which these joinpoints are

located is oblivious of the connectors that may be applied to it. Thus, these connectors allow augmenting a concern with other concerns that were not considered when it was designed, which facilitates independent evolution and reuse of these concerns. The joinpoints are not limited to single activities, but can also be groups of workflow nodes that form a single-entry single exit (SESE) fragment. This allows connecting concerns to parts of the workflow that were not modularized as a separate activity, and thus reduces coupling between concerns. In addition to the above connectors that are mainly influenced by aspect-oriented principles, the *replace connector* allows expressing that an existing activity in one concern should be implemented by executing another concern, in a way that minimizes dependencies between these concerns and thus facilitates their independent evolution and reuse. Thus, this connector is related to traditional component-based software development (CBSD). Next to these external connectors, *internal connectors* allow introducing additional control flow dependencies within parallel or conditional control structures in order to allow synchronizing parallel branches or switching alternative branches, which is useful in certain situations. By allowing every concern connection pattern to be natively specified using a dedicated connector, UNIFY enables specifying all of a concern's connection logic as a connector, while the concern's behavior is modularized in a CompositeActivity, which can be reused in other contexts by simply referencing it in a different connector.

Because UNIFY's connector mechanism constitutes a novel workflow modularization mechanism, we must ensure that the connector mechanism's semantics be precisely described, and that this semantics fits into the workflow community's existing formal tradition. Therefore, we provide a formalization of our approach that is compatible with existing research on this topic within the workflow community (van der Aalst, 1997, 1998a, 2000; van der Aalst et al., 2011), but also addresses the specific notion of connection patterns introduced by UNIFY. In order to formalize the aspect-oriented workflow concepts introduced by UNIFY, we employ two complementary formalisms. First, we augment the static description of UNIFY's workflows as provided by its base language and connector language meta-models with a static semantics for the weaving of UNIFY connectors using the Graph Transformation formalism (Rozenberg, 1997; Ehrig et al., 2006). This facilitates static reasoning over the applicability and effects of connectors, and can be used to implement a static weaver of UNIFY connectors. Second, we provide a semantics for the operational properties of workflows by defining a translation to Petri nets (Petri and Reisig, 2008), and subsequently extend this semantics to support the operational effects of connectors. This allows reasoning on the dynamics of UNIFY workflow compositions, and can be used to implement a dedicated workflow engine for UNIFY.

With PADUS and UNIFY, we have introduced two approaches that promote separation of concerns in workflow languages by offering an improved modularization mechanism. However, the abstractions offered by existing modularization approaches for workflows — and by PADUS and UNIFY — typically remain at the same level as the base workflow: concerns are implemented using the constructs of the base workflow language, which may not be ideally suited to expressing the concern in question in an efficient, elegant or natural way. Although aspect-oriented extensions improve separation of concerns, they introduce an additional layer of complexity that must be bridged in order to communicate about a workflow with domain experts. Inspired by the bene-

fits of domain-specific languages in general software engineering (van Deursen et al., 2000), we believe that a means of expressing workflow concerns using abstractions that are closer to the concerns' domains can facilitate expressing workflow concerns, and can improve communication with domain experts. Therefore, we develop a methodology for specifying *concern-specific languages* (CSLs) on top of UNIFY, and exemplify this methodology by developing two example CSLs, i.e., the *Access Control* and *Parental Control* CSLs, respectively. For each of these languages, we first identify the relevant domain concepts and relations, which results in a UML class diagram. Next, we specify an appropriate syntax that matches the concepts and relations identified in the previous step. Finally, we provide a mechanism by means of which artifacts expressed using the above syntax are translated into basic UNIFY artifacts by implementing a preprocessor for CSL artifacts. We provide example artifacts for both CSLs, and compare these with the corresponding UNIFY implementations in order to demonstrate the advantages of using concern-specific languages for workflow development.

Finally, we have implemented a proof-of-concept of PADUS in PROLOG, and have performed a qualitative validation of PADUS by showing how it can be used to add the billing concern to a multi-party conference call process. We have implemented a proof-of-concept of UNIFY in JAVA, and have instantiated the framework towards the WS-BPEL and BPMN workflow languages. We have performed a qualitative validation of UNIFY by comparing a number of concerns' UNIFY implementations with the equivalent implementations in standard WS-BPEL and AO4BPEL, respectively. We have performed an initial quantitative validation of UNIFY's performance and scalability by measuring the execution time of a number of example UNIFY workflows and comparing these measurements with measurements of the equivalent standard WS-BPEL workflows. In these experiments, UNIFY scales linearly with the amount of connectors applied to the workflow.

In conclusion of this section, we further summarize the above by reiterating our contributions:

1. We develop a novel, comprehensive approach for modularizing workflow concerns — UNIFY — which allows modularizing *all* workflow concerns using a single language construct, and is thus a *uniform* approach. At the heart of UNIFY lies a base language meta-model that is compatible with a wide range of existing workflow languages.

2. We introduce a number of *concern connection patterns* for workflows, which go beyond the classic aspect-oriented patterns by taking into account the specific properties of the workflow paradigm. In UNIFY, these patterns take the form of a *connector mechanism* that allows connecting independently specified workflow concerns according to each of the concern connection patterns. This connector mechanism is defined in terms of UNIFY's base language meta-model.

3. We enable the definition of *concern-specific languages* (CSLs) on top of UNIFY, which facilitate the definition of families of concerns using abstractions that are close to the concerns' domains. We provide a general methodology for CSL development and exemplify the methodology using two example CSLs.

4. We provide a precise semantics for UNIFY. We enable the verification of static properties and implementation of static weaving by defining a semantics for our concern connection patterns based on the Graph Transformation formalism, and enable the verification of operational properties and implementation of dynamic weaving by defining a semantics based on Petri nets.

5. We provide a proof-of-concept implementation of UNIFY, which has been implemented using JAVA, and which has been extended towards BPEL and BPMN. In order to promote compatibility with existing tool chains, UNIFY does not impose a modified workflow engine. We perform a qualitative validation of the expressiveness of UNIFY, and perform a quantitative validation of the performance and scalability of UNIFY's source code weaver.

8.2 Discussion and Future Work

Instantiating the UNIFY framework In our current implementation, we have instantiated the UNIFY framework towards the core concepts of WS-BPEL and BPMN, the de facto standards in executable workflows and graphical workflow modeling, respectively. This existing implementation can be extended with complete support for all language concepts of these languages, such as additional control flow patterns or workflow perspectives (e.g., exception handling). The existing implementation can also be extended with instantiations towards other workflow languages, such as YAWL. We have opted not to do this at this time, as we believe our current instantiations sufficiently illustrate the benefits of the UNIFY approach. Nevertheless, these extensions, as well as their impact on UNIFY's meta-models, constitute an interesting avenue of future work.

An environment for defining, composing, verifying and executing modular workflows Because neither PADUS nor UNIFY introduces incompatibilities with existing tool chains, either can be applied without using new tools to define, verify or execute workflows or workflow concerns. BPEL workflows or workflow concerns are currently specified using standard BPEL editors. The composition of workflow concerns is currently done in a text editor by specifying PADUS deployments or UNIFY compositions. Workflows are verified using standard tools, and are executed using standard engines. Nevertheless, we believe a dedicated environment, in which workflow concerns can be defined, composed, verified and executed according to the principles described in this dissertation could be beneficial to the adoption of our approach.

Leveraging our Petri net semantics in a dynamic weaver Although this dissertation provides a comprehensive semantics for the modularization concepts introduced by the UNIFY framework, we currently do not exploit the full potential of this semantics in our implementation. Because we were focusing on ensuring compatibility with existing tool chains, UNIFY's connector mechanism is currently implemented as a source code weaver that is invoked by our current workflow engine before executing the woven workflow using our Petri net semantics for UNIFY's base language. Moving away from the source code

weaver, towards a workflow engine that weaves UNIFY concerns at runtime according to our our Petri net semantics for UNIFY's connector mechanism, enables supporting novel features that cannot easily be implemented in our current implementation. Most notably, these features include *stateful aspects*: workflow concerns that execute only when a certain sequence of events within the base workflow has occured. One of our Master's students has extended PADUS with support for stateful aspects that are statically woven (Braem and Gheysels, 2007), but we consider dynamic weaving of stateful UNIFY concerns an interesting avenue for future research as well.

Tools for defining and using concern-specific languages In this dissertation, we have provided a methodology for developing concern-specific languages (CSLs) on top of the UNIFY framework, based on the different actors involved in workflow development and on current practice in the related research domain of domain-specific languages. We currently employ this methodology in combination with rudimentary tools such as a graphics editor for specifying UML class diagrams and a text editor for editing XML documents. The parser and code generator for a new CSL are currently implemented in JAVA from scratch. Although it is feasible to employ the above approach to develop new CSLs, we envision a CSL development tool that reduces the effort of developing new CSLs. This tool would include a UML editor to model the domain concepts and relations of a new CSL. The tool would then support creating a CSL syntax in a user-friendly way based on this UML model. By allowing to map CSL concepts to UNIFY workflow concepts, the tool would support semi-automatically generating a preprocessor that translates CSL artifacts to UNIFY artifacts. Note that different concerns expressed using the same CSL will typically have a similar structure. Therefore, one can conceive schemes that allow specifying concerns based on concern-specific *templates*. Such templates can be specified after defining the CSL syntax, and the definition of such templates would thus be integrated in the CSL development tool, while their use would be integrated in a CSL artifact specification tool. This tool support for CSL development is subject to future work.

Further validating the impact of UNIFY on workflow development In this dissertation, we have reported on an initial qualitative validation of UNIFY with respect to expressiveness based on our running example, and an initial quantitative validation of UNIFY with respect to performance and scalability of the UNIFY source code weaver based on synthetic example workflows. The scope of this validation can be extended by considering larger workflows as input for a qualitative study. We intend to collaborate with an industrial partner who can provide such a workflow, as large real-life workflows are typically not publicly available. We will also validate our concern-specific languages as part of this extended study. With regard to quantitative properties, further experiments are required in order to verify whether compositions remain scalable when other types of connectors are used than those in our current validation, when combinations of multiple types of connectors are used, or when connectors are applied to multiple joinpoints.

UNIFY in the cloud In our current implementation of UNIFY, we assume a workflow is executed by a traditional workflow engine, which is typically part of an application server hosted inside of an organization. However, within many organizations there is a trend towards viewing *Software as a Service*, and adopting *cloud computing* (Armbrust et al., 2010) as a means towards achieving this goal. Similar to other IT applications, moving workflows into the cloud could offer benefits such as the possibility of scaling the maximum number of concurrently running workflow instances up or down depending on the organization's current needs. There is already some interest in deploying workflows in the cloud,[1] but separation of concerns remains a relevant issue in this context. Therefore, it would be interesting to see how UNIFY could support the execution of workflows within the cloud, and thus enable *Workflow as a Service*.

A vehicle for research on modularization of workflows Finally, we believe UNIFY's focus on improving separation of concerns in workflows through novel modularization mechanisms, its reliance on general meta-models that can be instantiated towards existing concrete workflow languages, and its solid formal foundations provide a good basis for a vehicle for further research on modularization of workflows. Therefore, we aim to extend and expose the framework further in order to attract other researchers in the domain of workflow modularization towards using UNIFY to conduct their own experiments.

[1] For example, Amazon Simple Workflow Service (beta), cf. `http://aws.amazon.com/swf/`.

Appendix A

UNIFY Connector Syntax

This appendix lists the concrete syntax of the UNIFY connector language in Backus–Naur form.

```
<connector> ::= <external-connector>
              | <internal-connector>

<external-connector> ::= <before-connector>
                       | <after-connector>
                       | <replace-connector>
                       | <around-connector>
                       | <parallel-connector>
                       | <alternative-connector>
                       | <iterating-connector>

<internal-connector> ::= <synchronizing-connector>
                       | <switching-connector>

<before-connector> ::= "CONNECT" <activity>
                       "BEFORE" <pointcut>
                       ("" | "IF" <condition>)

<after-connector> ::= "CONNECT" <activity>
                      "AFTER" <pointcut>
                      ("" | "IF" <condition>)

<replace-connector> ::= "CONNECT" <activity>
                        "INSTEAD OF" <pointcut>

<around-connector> ::= "CONNECT" <activity>
                       "AROUND" <pointcut>
                       ("" | "IF" <condition>)
                       ("" | "PROCEEDING AT" <pointcut>)

<parallel-connector> ::= "CONNECT" <activity>
                         "PARALLEL TO" <pointcut>
                         ("" | "IF" <condition>)

<alternative-connector> ::= "CONNECT" <activity>
                            "ALTERNATIVE TO" <pointcut>
```

```
                              "IF" <condition>

<iterating-connector> ::= "CONNECT" <activity>
                          "ITERATING OVER" <pointcut>
                          "UNTIL" <condition>

<synchronizing-connector> ::= "CONNECT" <activity>
                              "IN" <pointcut>
                              "AND-SPLITTING AT" <pointcut>
                              "SYNCHRONIZING AT" <pointcut>

<switching-connector> ::= "CONNECT" <activity>
                          "IN" <pointcut>
                          "SWITCHING AT" <pointcut>
                          "IF" <condition>
                          "XOR-JOINING AT" <pointcut>
```

Appendix B

Soundness Proof for the After Connector

This appendix proves the soundness of UNIFY's after connector. The proof is analogous to the proof for the before connector presented in Section 5.4.4.1, and refers to the after connector's Petri net semantics as illustrated by Figure 5.20 (on page 130).

Option to Complete Let M be a marking in N_{Wc} such that $[i] \xrightarrow{\sigma} M$ (with $\sigma = t_1 \dots t_n$). Then we must prove that there exists a σ' in N_{Wc} such that $M \xrightarrow{\sigma'} [o]$. Let j (resp. k) be the largest position in σ such that $t_j = t_a$ ($t_k = t_b$)

1. If $k \geq j$ (i.e., the advice is inactive in M)

   - Since either the advice has never been activated in σ ($j = k = 0$) or it has been activated but has finished ($k > j > 0$), and since the advice respects **proper completion**, we know that $\forall p \in P_a : M(p) = 0$
   - By Property A, $M\big|_{P_x}$ is reachable in N_{Wx}
   - Since N_{Wx} respects **option to complete**, there exists a $\overline{\sigma}$ in N_{Wx} such that $M\big|_{P_x} \xrightarrow{\overline{\sigma}} [o]$
   - Let θ be the sequence of transitions obtained from $\overline{\sigma}$ by replacing each occurrence of t_b by $t_a \sigma_a t_b$, where σ_a is a sequence of transitions of N_{Wa} such that $[p'_s] \xrightarrow{\sigma_a} [p'_e]$ (such a σ_a always exists because N_{Wa} respects **option to complete** and **proper completion**).

     That is if $\overline{\sigma} = \sigma_1 t_b \sigma_2 t_b \dots t_b \sigma_k$ where all σ_i do not contain t_b, then $\theta = \sigma_1 t_a \sigma_a t_b \sigma_2 t_a \sigma_a t_b \dots t_a \sigma_a t_b \sigma_k$
   - It is easy to show that θ is fireable in N_{Wc} (due to Properties A, B, C) and that it reaches $[o]$

2. If $k < j$ (i.e., the advice is active in M)

- By Property A, $M\big|_{P_a}$ is reachable in N_{Wa}
- Since N_{Wa} respects **option to complete**, there exists a sequence σ' in N_{Wa} such that $M\big|_{P_a} \xrightarrow{\sigma'} [p'_e]$
- By Property B, σ' is fireable in N_{Wc}, and

$$M \xrightarrow{\sigma'} M' \text{ implies } \begin{cases} M'(p'_e) = 1 & \\ M'(p) = 0 & \forall\, p \in P_a \setminus \{p'_e\} \\ M'(p) = M(p) & \forall\, p \in P_x \end{cases}$$

- Thus, t_b is fireable from M' and moves the token from p'_e to p_b
- From there, we continue as in (1)

Proper Completion Let M be a marking that is reachable in N_{Wc} such that $M \geq [o]$. Then we must prove that $M = [o]$. Assume $[i] \xrightarrow{\sigma} M$, with $\sigma = \sigma_1 t_a \theta_1 t_b \sigma_2 t_a \theta_2 t_b \ldots \sigma_n t_a \theta_n t_b \sigma_{n+1}$, every θ_i giving rise to marking M_i, and every subsequent t_b giving rise to marking M'_i

- $\forall i : M_i \geq p'_e$. Since N_{Wa} respects **proper completion**, $M_i\big|_{P_a} = p'_e$, thus $\forall p \in P_a$: $M'_i(p) = 0$, $\forall p \in P_x \setminus \{p_b\} : M'_i(p) = M_i(p)$, and $M'_i(p_b) = 1$
- By Property A, $M'_n\big|_{P_x}$ is reachable in N_{Wx} and σ_{n+1} is fireable in N_{Wx} (because $\forall p \in P_a : M'_n(p) = 0$), and $M'_n\big|_{P_x} \xrightarrow{\sigma_{n+1}} M\big|_{P_x}$
- By hypothesis, $M \geq [o]$, and thus $M\big|_{P_x} \geq [o]$ because $o \in P_x$
- By hypothesis, N_{Wx} respects **proper completion** such that $M\big|_{P_x} = [o]$
- Moreover, since $\forall p \in P_a : M'_n(p) = 0$ and σ_{n+1} does not contain t_a, $\forall p \in P_a$: $M(p) = 0$
- Therefore, $M = [o]$

No Dead Transitions Given a transition $t \in T_c$, we must find a sequence σ such that $[i] \xrightarrow{\sigma} M \xrightarrow{t} M'$

1. If $t \in T_x$
   - Since N_{Wx} respects **no dead transitions**, we know that there exists a θ such that $[i] \xrightarrow[N_{Wx}]{\theta} M \xrightarrow{t} M'$
   - From θ, we can build $\overline{\theta}$ by replacing each occurrence of t_b by $t_a \overline{\sigma} t_b$ where $\overline{\sigma}$ is a sequence of the advice such that $[p'_s] \xrightarrow[N_{Wa}]{\overline{\sigma}} [p'_e]$, which exists because N_{Wa} respects **option to complete**.

- By Properties B and C, $\overline{\theta}$ is fireable in N_{Wc} and reaches $\overline{M}$ such that $\overline{M}\big|_{P_x} = M$

2. If $t \in T_a$
   - We know that t_b is not dead in N_{Wx} because it respects **no dead transitions**.
   - By the same reasoning as in case 1, we can build $\overline{\theta}$ such that $[i] \xrightarrow[N_{Wc}]{\overline{\theta}} M'' \xrightarrow{t_a} M'''$ with $M'''(p'_s) = 1$
   - Since t is not dead in N_{Wa} because it respects **no dead transitions**, there exists a τ such that $[p'_s] \xrightarrow[N_{Wa}]{\tau} M \xrightarrow{t} M'$
   - By Properties B and C, $\overline{\theta}\, t_a \tau$ is fireable in N_{Wc} and reaches M such that $M \xrightarrow{t} M'$

Q.E.D.

Appendix C

Access Control and Parental Control CSL Syntax

This appendix lists the concrete syntax of the Access Control and Parental Control CSLs, which is defined using XML SCHEMA.

C.1 Access Control CSL Syntax

```
<?xml version="1.0" ?>
<xs:schema elementFormDefault="qualified"
     targetNamespace="http://unify-framework.org/DSLs/AccessControl"
     xmlns="http://unify-framework.org/DSLs/AccessControl"
     xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AccessControlConcern">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="DefaultPermission" type="DefaultPermissionType"
             minOccurs="0" maxOccurs="1" />
        <xs:element name="DefaultAction" type="DefaultActionType" minOccurs="0"
             maxOccurs="1" />
        <xs:element name="Role" type="RoleType" minOccurs="1"
             maxOccurs="unbounded" />
        <xs:element name="User" type="UserType" minOccurs="1"
             maxOccurs="unbounded" />
        <xs:element name="DefaultUser" type="DefaultUserType" minOccurs="0"
             maxOccurs="1" />
        <xs:element name="UsernameVariable" type="UsernameVariableType"
             minOccurs="0" maxOccurs="1" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" />
    </xs:complexType>
  </xs:element>

  <xs:complexType name="DefaultPermissionType">
    <xs:attribute name="permission" use="required">
      <xs:simpleType>
```

```
      <xs:restriction base="xs:string">
        <xs:pattern value="Allow|Deny"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>

<xs:complexType name="DefaultActionType">
  <xs:attribute name="action" type="ActionType" use="required" />
</xs:complexType>

<xs:simpleType name="ActionType">
  <xs:restriction base="xs:string">
    <xs:pattern value="RaiseError|Skip"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="RoleType">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="Allow" type="AllowType" />
    <xs:element name="Deny" type="DenyType" />
  </xs:choice>
  <xs:attribute name="name" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="AllowType">
  <xs:attribute name="activity" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="DenyType">
  <xs:attribute name="activity" type="xs:string" use="required" />
  <xs:attribute name="action" type="ActionType" />
</xs:complexType>

<xs:complexType name="DefaultUserType">
  <xs:sequence>
    <xs:element name="UserRole" type="UserRoleType" minOccurs="0"
        maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="UserRoleType">
  <xs:attribute name="role" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="UserType">
  <xs:sequence>
    <xs:element name="UserRole" type="UserRoleType" minOccurs="0"
        maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required" />
  <xs:attribute name="password" type="xs:string" />
</xs:complexType>

<xs:complexType name="UsernameVariableType">
  <xs:attribute name="expression" type="xs:string" use="required" />
```

```
    </xs:complexType>

  </xs:schema>
```

C.2 Parental Control CSL Syntax

```
<?xml version="1.0" ?>
<xs:schema elementFormDefault="qualified"
     targetNamespace="http://unify-framework.org/DSLs/ParentalControl"
     xmlns="http://unify-framework.org/DSLs/ParentalControl"
     xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="ParentalControlConcern">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Policies" type="PoliciesType" minOccurs="1"
             maxOccurs="unbounded" />
        <xs:element name="Child" type="ChildType" minOccurs="0"
             maxOccurs="unbounded" />
        <xs:element name="AgeVariable" type="AgeVariableType" minOccurs="1"
             maxOccurs="1" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" />
    </xs:complexType>
  </xs:element>

  <xs:complexType name="PoliciesType">
    <xs:sequence>
      <xs:element name="Filter" type="FilterType" minOccurs="0"
           maxOccurs="unbounded" />
      <xs:element name="Deny" type="DenyType" minOccurs="0" maxOccurs="unbounded"
           />
      <xs:element name="ReferToParent" type="ReferToParentType" minOccurs="0"
           maxOccurs="unbounded" />
      <xs:element name="Monitor" type="MonitorType" minOccurs="0"
           maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="youngerThan" type="xs:integer" />
  </xs:complexType>

  <xs:complexType name="FilterType">
    <xs:sequence>
      <xs:element name="Exclude" type="ExcludeType" minOccurs="1"
           maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="activity" type="xs:string" />
    <xs:attribute name="resultVariableExpression" type="xs:string" />
  </xs:complexType>

  <xs:complexType name="ExcludeType">
    <xs:attribute name="property" type="xs:string" />
    <xs:attribute name="value" type="xs:string" />
  </xs:complexType>

  <xs:complexType name="DenyType">
    <xs:attribute name="activity" type="xs:string" />
```

```
  </xs:complexType>

  <xs:complexType name="ReferToParentType">
    <xs:attribute name="activity" type="xs:string" />
    <xs:attribute name="usernameVariableExpression" type="xs:string" />
  </xs:complexType>

  <xs:complexType name="MonitorType">
    <xs:attribute name="activity" type="xs:string" />
    <xs:attribute name="usernameVariableExpression" type="xs:string" />
  </xs:complexType>

  <xs:complexType name="ChildType">
    <xs:sequence>
      <xs:element name="Parent" type="ParentType" minOccurs="1"
           maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" />
  </xs:complexType>

  <xs:complexType name="ParentType">
    <xs:attribute name="name" type="xs:string" />
  </xs:complexType>

  <xs:complexType name="AgeVariableType">
    <xs:attribute name="expression" type="xs:string" />
  </xs:complexType>

</xs:schema>
```

Appendix D

Generated Parental Control Concerns

This appendix lists the WS-BPEL concerns generated by the Parental Control CSL code generator for the example concerns of Section 6.5.

D.1 Generated Filtering Concern

```
<scope name="GeneratedFilteringActivity"
    xmlns:bed="http://back_end.order_books.examples.unify_framework.org/xsd"
    xmlns:fbe="http://back_end.filtering.examples.unify_framework.org/"
    xmlns:fbed="http://back_end.filtering.examples.unify_framework.org/xsd">
  <partnerLinks>
    <partnerLink name="FilteringBackEndPartnerLink"
        partnerLinkType="fbe:FilteringBackEndPartnerLinkType"
        partnerRole="me" />
  </partnerLinks>
  <variables>
    <variable messageType="fbe:filterRequest" name="backEndFilterInput" />
    <variable messageType="fbe:filterResponse" name="backEndFilterOutput" />
  </variables>
  <sequence>
    <if>
      <condition>$user/bed:age&lt;18</condition>
      <sequence>
        <assign name="PrepareFilteringInput">
          <copy>
            <from>
              <literal>
                <filter xmlns="http://back_end.filtering.examples
                    .unify_framework.org/xsd">
                  <data>
                    <books />
                  </data>
                  <criterion property="Rating" value="Mature" />
                  <criterion property="Genre" value="Horror" />
                </filter>
              </literal>
            </from>
```

```
            <to part="parameters" variable="backEndFilterInput" />
          </copy>
          <copy>
            <from variable="books" />
            <to part="parameters" variable="backEndFilterInput">
              <query>//fbed:filter/fbed:data/fbed:books</query>
            </to>
          </copy>
        </assign>
        <invoke inputVariable="backEndFilterInput"
            name="InvokeFilteringService" operation="filter"
            outputVariable="backEndFilterOutput"
            partnerLink="FilteringBackEndPartnerLink"
            portType="fbe:FilteringBackEndPortType" />
        <assign name="ProcessFilteringOutput">
          <copy>
            <from part="parameters" variable="backEndFilterOutput">
              <query>//fbed:filterResponse/fbed:return/fbed:books</query>
            </from>
            <to variable="books" />
          </copy>
        </assign>
      </sequence>
      <else>
        <sequence>
          <empty name="DoNothing" />
        </sequence>
      </else>
    </if>
  </sequence>
</scope>
```

D.2 Generated Deny Usage Concern

```
<scope name="GeneratedDenyUsageActivity">
  <partnerLinks />
  <variables />
  <sequence>
    <empty name="DoNothing" />
  </sequence>
</scope>
```

D.3 Generated Refer Usage Concern

```
<scope name="GeneratedReferUsageActivity">
  <partnerLinks />
  <variables>
    <variable name="UsernameBackup" type="xsd:string" />
    <variable element="d:dictionary" name="ParentsDb"
      xmlns:d="http://unify-framework.org/Util/Dictionary" />
  </variables>
  <sequence>
    <assign name="InitializeParentsDatabase">
      <copy>
        <from>
```

```
          <literal>
            <dictionary xmlns="http://unify-framework.org/Util/Dictionary">
              <key key="suzy">
                <value value="george" />
              </key>
            </dictionary>
          </literal>
        </from>
        <to variable="ParentsDb" />
      </copy>
    </assign>
    <assign name="BackupUsername">
      <copy>
        <from>$user/bed:username</from>
        <to variable="UsernameBackup" />
      </copy>
    </assign>
    <assign name="ReplaceUsername">
      <copy>
        <from xmlns:d="http://unify-framework.org/Util/Dictionary">
          $ParentsDb/d:key[@key=$UsernameBackup]/d:value/@value
        </from>
        <to>$user/bed:username</to>
      </copy>
    </assign>
    <scope name="SpecifyPaymentInfo">
      <!-- This is a copy of the joinpoint activity -->
    </scope>
    <assign name="RestoreUsername">
      <copy>
        <from>$UsernameBackup</from>
        <to>$user/bed:username</to>
      </copy>
    </assign>
  </sequence>
</scope>
```

D.4 Generated Monitoring Concern

```
<scope name="GeneratedMonitoringActivity"
    xmlns:bed="http://back_end.order_books.examples.unify_framework.org/xsd"
    xmlns:mbe="http://back_end.monitoring.examples.unify_framework.org/"
    xmlns:mbed="http://back_end.monitoring.examples.unify_framework.org/xsd">
  <partnerLinks>
    <partnerLink name="MonitoringBackEndPartnerLink"
        partnerLinkType="mbe:MonitoringBackEndPartnerLinkType"
        partnerRole="me" />
  </partnerLinks>
  <variables>
    <variable messageType="mbe:monitorRequest" name="backEndMonitorInput" />
    <variable messageType="mbe:monitorResponse" name="backEndMonitorOutput" />
  </variables>
  <sequence>
    <if>
      <condition>$user/bed:age&lt;18</condition>
      <sequence>
```

```
          <assign name="PrepareMonitoringInfo">
            <copy>
              <from>
                <literal>
                  <monitor xmlns="http://back_end.monitoring.examples
                      .unify_framework.org/xsd">
                    <message>Underage user is executing activity
                        OrderBooks.SelectBooks.Confirm</message>
                    <username />
                  </monitor>
                </literal>
              </from>
              <to part="parameters" variable="backEndMonitorInput" />
            </copy>
            <copy>
              <from>$user/bed:username/text()</from>
              <to part="parameters" variable="backEndMonitorInput">
                <query>//mbed:monitor/mbed:username/text()</query>
              </to>
            </copy>
          </assign>
          <invoke inputVariable="backEndMonitorInput"
              name="InvokeMonitoringService" operation="monitor"
              outputVariable="backEndMonitorOutput"
              partnerLink="MonitoringBackEndPartnerLink"
              portType="mbe:MonitoringBackEndPortType" />
        </sequence>
        <else>
          <sequence>
            <empty name="DoNothing" />
          </sequence>
        </else>
      </if>
    </sequence>
  </scope>
```

Bibliography

Active Endpoints. ActiveBPEL, version 2.1, June 2006. URL `http://www.activebpel.org/`. Cited on pages 19 and 49.

Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, Heidelberg, Germany, 2004. ISBN 978-3-540-44008-6. Cited on page 16.

Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services, version 1.1, May 2003. URL `http://www.ibm.com/developerworks/library/specification/ws-bpel/`. Cited on pages 1 and 17.

Apache Software Foundation. Apache log4j, version 1.2, May 2002. URL `http://http://logging.apache.org/log4j/`. Cited on page 25.

Apache Software Foundation. Orchestration Director Engine (ODE), version 1.3.3, August 2009. URL `http://ode.apache.org/`. Cited on pages 19 and 49.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010. Cited on page 201.

Ali Arsanjani, Brent Hailpern, Joanne Martin, and Peri Tarr. Web services: Promises and compromises. *ACM Queue*, 1(1):48–58, March 2003. Cited on pages 2, 31, and 35.

Uwe Aßmann. *Invasive Software Composition*. Springer, Heidelberg, Germany, 2003. Cited on pages 97 and 98.

Jon Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986. Cited on page 152.

Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001. Cited on page 98.

Eric Bodden. Concern-specific languages and their implementation with abc. In *Proceedings of the 3rd International Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT 2005)*, Chicago, IL, USA, March 2005. Cited on page 153.

Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP), version 1.1. W3C Note 08 May 2000, World Wide Web Consortium, May 2000. URL `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`. Cited on page 17.

Mathieu Braem and Dimitri Gheysels. History-based aspect weaving for WS-BPEL using Padus. In *Proceedings of the 5th IEEE European Conference on Web Services (ECOWS 2007)*, pages 159–167, Halle (Saale), Germany, November 2007. Cited on pages 81 and 200.

Mathieu Braem, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, and Viviane Jonckers. Concern-specific languages in a visual web service creation environment. In *Proceedings of the 2nd International Workshop on Aspect-based and Model-based Separation of Concerns in Software Systems (ABMB 2006)*, volume 163(2) of *Electronic Notes in Theoretical Computer Science*, pages 3–17, Bilbao, Spain, July 2006a. Elsevier. Cited on pages 7, 50, 153, and 164.

Mathieu Braem, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, and Viviane Jonckers. Guiding service composition in a visual service creation environment. In *Proceedings of the 4th IEEE European Conference on Web Services (ECOWS 2006)*, pages 13–22, Zürich, Switzerland, December 2006b. IEEE Computer Society. Cited on pages 7 and 50.

Mathieu Braem, Kris Verlaenen, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, Eddy Truyen, Wouter Joosen, and Viviane Jonckers. Isolating process-level concerns using Padus. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 113–128, Vienna, Austria, September 2006c. Springer. Cited on pages 3, 5, 7, 153, and 164.

Johan Brichau and Michael Haupt. Survey of aspect-oriented languages and execution models. AOSD-Europe Deliverable D12: AOSD-Europe-VUB-01, AOSD-Europe Network of Excellence, May 2005. URL `http://www.aosd-europe.net/deliverables/d12.pdf`. Cited on pages 27, 38, and 79.

R. Campbell and A. Habermann. The specification of process synchronisation by path expressions. In *Proceedings of an International Symposium on Operating Systems*, pages 89–102, April 1974. Cited on page 53.

Anis Charfi and Mira Mezini. Aspect-oriented web service composition with AO4BPEL. In *Proceedings of the 2nd European Conference on Web Services (ECOWS 2004)*, volume 3250 of *Lecture Notes in Computer Science*, pages 168–182, Erfurt, Germany, September 2004. Springer. Cited on pages 2, 31, 32, 35, and 37.

Anis Charfi and Mira Mezini. AO4BPEL: An aspect-oriented extension to BPEL. *World Wide Web*, 10(3):309–344, September 2007. Cited on page 32.

Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL), version 1.1. W3C Note 15 March 2001, World Wide Web Consortium, March 2001. URL `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`. Cited on page 17.

María Agustina Cibrán. *Connecting High-Level Business Rules with Object-Oriented Applications*. PhD thesis, Vrije Universiteit Brussel, System and Software Engineering Lab, Brussels, Belgium, June 2007. Cited on page 166.

Carine Courbis and Anthony Finkelstein. Towards an aspect weaving BPEL engine. In *Proceedings of the 3rd International Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2004)*, Lancaster, United Kingdom, March 2004. Cited on pages 2, 31, 32, and 35.

Carine Courbis and Anthony Finkelstein. Towards aspect weaving applications. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 69–77, St. Louis, MO, USA, May 2005a. ACM Press. Cited on pages 2, 32, 33, and 37.

Carine Courbis and Anthony Finkelstein. Weaving aspects into web service orchestrations. In *Proceedings of the 3rd IEEE International Conference on Web Services (ICWS 2005)*, pages 69–77, Orlando, FL, USA, July 2005b. IEEE Computer Society. Cited on pages 32 and 33.

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, USA, 2000. ISBN 978-0-201-30977-5. Cited on page 153.

Thomas H. Davenport and James E. Short. The new industrial engineering: Information technology and business process redesign. *Sloan Management Review*, 31(4):11–27, Summer 1990. Cited on page 10.

Bruno De Fraine. *Language Facilities for the Deployment of Reusable Aspects*. PhD thesis, Vrije Universiteit Brussel, Software Languages Lab, Brussels, Belgium, June 2009. Cited on page 29.

Kris De Volder. Aspect-oriented logic meta programming. In *Proceedings of the Aspect Oriented Programming Workshop at ECOOP 1998*, Brussels, Belgium, June 1998. Cited on page 39.

Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni, editors. *Prolog: The Standard*. Springer, Heidelberg, Germany, 1996. ISBN 978-3-540-59304-1. Cited on page 40.

Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12):1281–1294, November 2008. Cited on page 102.

Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972. Cited on page 97.

Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*. Springer, New York, NY, USA, 1982. ISBN 0-387-90652-5. Originally published as EWD447, August 1974. Cited on pages 1, 23, and 58.

Ming Dong and F. Frank Chen. Petri net-based workflow modelling and analysis of the integrated manufacturing business processes. *The International Journal of Advanced Manufacturing Technology*, 26(9):1163–1172, 2005. Cited on page 31.

Eclipse Foundation. Eclipse BPEL Designer, version 0.5.0, June 2011. URL `http://www.eclipse.org/bpel/`. Cited on page 18.

Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In *Proceedings of the 2nd International Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2004. Cited on page 112.

Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, Heidelberg, Germany, 2006. ISBN 978-3-540-31187-4. Cited on pages 5, 102, and 197.

Johan Fabry. *Modularizing Advanced Transaction Management: Tackling Tangled Aspect Code*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Lab, Brussels, Belgium, 2005. Cited on page 153.

David C. Fallside and Priscilla Walmsley. XML Schema part 0: Primer, second edition. W3C Recommendation 28 October 2004, World Wide Web Consortium, October 2004. URL `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`. Cited on page 17.

Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical Report 01.12, Research Institute for Advanced Computer Science, May 2001. Cited on page 27.

Alexander Förster, Gregor Engels, Tim Schattkowsky, and Ragnhild Van Der Straeten. Verification of business process quality constraints based on visual process patterns. In *Proceedings of the 1st IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 197–208, Shanghai, China, June 2007. IEEE Computer Society. Cited on page 81.

Dimitri Gheysels. Implementation of stateful aspects in Padus. Master's thesis, Vrije Universiteit Brussel, System and Software Engineering Lab, June 2007. URL `http://soft.vub.ac.be/~njonchee/theses/thesis_dimitri.pdf`. Cited on page 81.

Frank B. Gilbreth. Process charts: First steps in finding the one best way to do work. *ASME Transactions*, 43, 1922. Cited on page 9.

Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Indianapolis, IN, USA, 2004. ISBN 978-0-471-20284-4. Cited on page 153.

Sebastian Günther. *Development and Utilization of Internal Domain-Specific Languages*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, Magdeburg, Germany, 2010. Cited on page 153.

Sebastian Günther, Thomas Cleenewerck, and Viviane Jonckers. Software variability: The design space of configuration languages. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 157–164, Leipzig, Germany, January 2012. ACM Press. Cited on page 154.

Hugo Haas and Allen Brown. Web services glossary. W3C Working Group Note 11 February 2004, World Wide Web Consortium, February 2004. URL `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/`. Cited on page 16.

Wilke Havinga, Lodewijk Bergmans, and Mehmet Aksit. A model for composable composition operators: Expressing object and aspect compositions with first-class operators. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 145–156, Saint-Malo, France, March 2010. ACM Press. Cited on page 98.

Jan Heering. Application software, domain-specific languages, and language design assistants. Technical Report SEN-R0010, Centrum Wiskunde & Informatica, Amsterdam, Netherlands, May 2000. URL `http://oai.cwi.nl/oai/asset/4444/04444D.pdf`. Cited on page 152.

Erik Hilsdale, Jim Hugunin, Mik Kersten, Gregor Kiczales, and Jeffrey Palm. Aspect-oriented programming in Java with AspectJ. Presented at the O'Reilly Conference on Enterprise Java, March 2001. Cited on pages xv and 26.

Jendrik Johannes. *Component-Based Model-Driven Software Development*. PhD thesis, Technische Universität Dresden, December 2010. Cited on page 98.

Niels Joncheere. The Service Creation Environment: A telecom case study. In *Proceedings of the 5th International Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT 2007)*, Vancouver, BC, Canada, March 2007. Cited on page 7.

Niels Joncheere and Ragnhild Van Der Straeten. Semantics of the Unify composition mechanism. Technical Report SOFT-TR-2011.04.15, Vrije Universiteit Brussel, Software Languages Lab, Brussels, Belgium, April 2011a. URL `http://soft.vub.ac.be/~njonchee/publications/TR20110415.pdf`. Cited on page 7.

Niels Joncheere and Ragnhild Van Der Straeten. Uniform modularization of workflow concerns using Unify. In *Proceedings of the 4th International Conference on Software Language Engineering (SLE 2011)*, volume 6940 of *Lecture Notes in Computer Science*, pages 77–96, Braga, Portugal, July 2011b. Springer. Cited on pages 3, 5, and 7.

Niels Joncheere, Wim Vanderperren, Mathieu Braem, and Ragnhild Van Der Straeten. Supporting user-friendly composition of web services in the Eclipse platform. In *Proceedings of the Eclipse Technology Exchange Workshop at ECOOP 2006*, Nantes, France, July 2006. Cited on page 7.

Niels Joncheere, Dirk Deridder, Ragnhild Van Der Straeten, and Viviane Jonckers. A framework for advanced modularization and data flow in workflow systems. In *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC 2008)*, volume 5364 of *Lecture Notes in Computer Science*, pages 592–598, Sydney, NSW, Australia, December 2008. Springer. Cited on pages 5 and 7.

Niels Joncheere et al. The Padus weaver, build 2009.09.15, September 2009. URL `http://www.padus.org/`. Cited on page 49.

Niels Joncheere et al. The Unify framework, 2012. URL `http://www.unify-framework.org/`. Cited on page 167.

Diane Jordan, John Evdemon, et al. Web Services Business Process Execution Language, version 2.0. OASIS Standard, OASIS, April 2007. URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`. Cited on pages 1 and 17.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer. Cited on pages 2, 24, 25, 59, and 79.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354, Budapest, Hungary, June 2001. Springer. Cited on pages 27, 41, and 82.

Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Christoph Bussler. On structured workflow modelling. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000)*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445, Stockholm, Sweden, June 2000. Springer. Cited on pages 15 and 62.

Marcello La Rosa, Arthur H. M. ter Hofstede, Petia Wohed, Hajo A. Reijers, Jan Mendling, and Wil M. P. van der Aalst. Managing process model complexity via concrete syntax modifications. *IEEE Transactions on Industrial Informatics*, 7(2):255–265, May 2011a. Cited on pages 11 and 68.

Marcello La Rosa, Petia Wohed, Jan Mendling, Arthur H. M. ter Hofstede, Hajo A. Reijers, and Wil M. P. van der Aalst. Managing process model complexity via abstract syntax modifications. *IEEE Transactions on Industrial Informatics*, 7(4):614–629, November 2011b. Cited on pages 2, 11, and 68.

Frank Leymann and Dieter Roller. Workflow-based applications. *IBM Systems Journal*, 36(1):102–123, 1997. Cited on page 31.

Karl Lieberherr, David H. Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University, College of Computer Science, Boston, MA, USA, March 1999. URL `http://www.cs.virginia.edu/~lorenz/papers/reports/NU-CCS-99-01.html`. Cited on page 83.

Niels Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0 and its compiler BPEL2oWFN. In *Proceedings of the 4th International Workshop on Web Services and Formal Methods (WS-FM 2007)*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91, Brisbane, QLD, Australia, September 2007. Springer. Cited on pages 4, 60, and 102.

Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, College of Computer Science, Boston, MA, USA, November 1997. Cited on page 153.

Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, Xerox Palo Alto Research Center, Palo Alto, CA, USA, February 1997. URL `http://www2.parc.com/csl/groups/sda/publications/papers/PARC-AOP-D97/for-web.pdf`. Cited on page 153.

Christophe Loridan and Jordi Anguela Rosell. BONITA: Workflow patterns support. Technical report, Bull R&D, May 2006. URL `http://www.workflowpatterns.com/vendors/documentation/bonita_patterns.pdf`. Cited on page 12.

D. Luckham, J. Kenney, L. Augustin, D. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21, 1995. Cited on page 53.

Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10): 1039–1065, August 2006. Cited on page 15.

C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, and Rebekah Metz. Reference model for service oriented architecture, version 1.0. OASIS Standard, OASIS, October 2006. URL `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`. Cited on page 16.

Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 2–28, Darmstadt, Germany, July 2003. Springer. Cited on page 25.

Jan Mendling, Gustaf Neumann, and Wil M. P. van der Aalst. Understanding the occurrence of errors in process models based on metrics. In *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS 2007)*, pages 113–130, Vilamoura, Portugal, 2007. Springer. Cited on page 101.

Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2006. Cited on pages 112 and 113.

Nataliya Mulyar. Pattern-based evaluation of Oracle-BPEL (v.10.1.2). BPM Center Report BPM-05-24, BPM Center, 2005. URL `http://www.workflowpatterns.com/vendors/documentation/Oracle_BPEL_v.10.1.2.pdf`. Cited on page 12.

James M. Neighbors. *Software Construction using Components*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, 1980. Cited on page 152.

Noesis Solutions. OPTIMUS, version 5.2, 2006. URL `http://www.noesissolutions.com/`. Cited on page 15.

Object Management Group. Object Constraint Language, version 2.0, May 2006. URL `http://www.omg.org/technology/documents/formal/ocl.htm`. Cited on page 62.

Object Management Group. Unified Modeling Language, superstructure, version 2.1.2, November 2007. URL `http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/`. Cited on page 62.

Object Management Group. Business Process Model and Notation, version 2.0, January 2011. URL `http://www.omg.org/spec/BPMN/2.0/`. Cited on page 20.

Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, June 2004. Cited on page 15.

Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, July 2007. Cited on page 102.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. Cited on pages 1, 23, 58, and 191.

Carl Adam Petri and Wolfgang Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008. Cited on pages 6, 101, and 197.

Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In *Term Graph Rewriting*, pages 201–213. Wiley, 1993. Cited on page 112.

Hajo A. Reijers and Jan Mendling. Modularity in process models: Review and effects. In *Proceedings of the 6th International Conference on Business Process Management (BPM 2008)*, volume 5240 of *Lecture Notes in Computer Science*, pages 20–35, Milan, Italy, September 2008. Springer. Cited on page 31.

Ralf H. Reussner. Automatic component protocol adaptation with the CoCoNut tool suite. *Future Generation Computer Systems*, 19(5):627–639, 2003. Cited on page 53.

Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, River Edge, NJ, USA, 1997. ISBN 978-981-02-2884-2. Cited on pages 5, 102, 103, and 197.

Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns. QUT Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, QLD, Australia, 2004a. URL `http://www.workflowpatterns.com/documentation/documents/data_patterns%20BETA%20TR.pdf`. Cited on pages 1, 11, 65, and 67.

Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow resource patterns. BETA Working Paper 127, Eindhoven University of Technology, Eindhoven, Netherlands, 2004b. URL `http://www.workflowpatterns.com/documentation/documents/Resource%20Patterns%20BETA%20TR.pdf`. Cited on pages 1, 11, and 68.

Nick Russell, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and Nataliya Mulyar. Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPM Center, 2006a. URL `http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf`. Cited on pages xviii, 1, 10, 11, 12, 22, 63, 65, 67, and 170.

Nick Russell, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. Exception handling patterns in process-aware information systems. BPM Center Report BPM-06-04, BPM Center, 2006b. URL `http://workflowpatterns.com/documentation/documents/BPM-06-04.pdf`. Cited on pages 1, 11, 68, and 170.

Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proceedings of the 5th ACM International Workshop on Role-based Access Control (RBAC 2000)*, pages 47–63, Berlin, Germany, July 2000. Cited on page 155.

Alec Sharp and Patrick McDermott. *Workflow Modeling: Tools for Process Improvement and Application Development*. Artech House, Norwood, MA, USA, 2001. ISBN 978-1-58053-021-7. Cited on page 31.

Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, USA, 1996. ISBN 978-0-13-182957-2. Cited on pages 4, 29, and 83.

Adam Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations*. W. Strahan and T. Cadell, London, United Kingdom, 1776. Cited on page 9.

Davy Suvée and Wim Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 21–29, Boston, MA, USA, March 2003. ACM Press. Cited on pages 4, 29, and 83.

Davy Suvée, Bruno De Fraine, and Wim Vanderperren. A symmetric and unified approach towards combining aspect-oriented and component-based software development. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, volume 4063 of *Lecture Notes in Computer Science*, pages 114–122, Västerås, Sweden, June 2006. Springer. Cited on pages 3, 5, 82, and 196.

Gabriele Taentzer et al. The Attributed Graph Grammar system: A development environment for attributed graph transformation systems, version 1.6.4, 2009. Cited on pages 103 and 113.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 107–119, Los Angeles, CA, USA, May 1999. IEEE Computer Society. Cited on pages 2, 3, 5, 24, 82, and 196.

Scott Thibault, Renaud Marlet, and Charles Consel. A domain-specific language for video device drivers: From design to implementation. In *Proceedings of the 1st International Conference on Domain-Specific Languages (DSL 1997)*, pages 11–26, October 1997. Cited on page 152.

Ivana Trickovic. Modularization and reuse in WS-BPEL. Sap community contribution, SAP AG, October 2005. URL `http://scn.sap.com/docs/DOC-1297`. Cited on page 1.

J. van den Bos and C. Laffra. PROCOL: A concurrent object-oriented language with protocols delegation and constraints. *Acta Informatica*, 28:511–538, June 1991. Cited on page 53.

Wil M. P. van der Aalst. Verification of workflow nets. In *Proceedings of the 18th International Conference on the Application and Theory of Petri Nets (ICATPN 1997)*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426, Toulouse, France, June 1997. Springer. Cited on pages 5, 101, and 197.

Wil M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, February 1998a. Cited on pages 5, 22, 101, 117, and 197.

Wil M. P. van der Aalst. Three good reasons for using a Petri-net-based workflow management system. In Toshiro Wakayama, Srikanth Kannapan, Chan Meng Khoong, Shamkant Navathe, and JoAnne Yates, editors, *Information and Process Integration in Enterprises*, volume 428 of *The Kluwer International Series in Engineering and Computer Science*, chapter 10, pages 161–182. Kluwer Academic Publishers, Boston, MA, USA, 1998b. ISBN 978-1-4615-5499-8. Cited on pages xvi, 22, 101, 117, 118, 124, and 149.

Wil M. P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In Wil M. P. van der Aalst, Jörg Desel, and Andreas Oberweis,

editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer, 2000. ISBN 3-540-67454-3. Cited on pages 5, 101, 119, 138, and 197.

Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, June 2005. Cited on pages xv, 1, 4, 11, 22, 24, 60, 102, 117, and 170.

Wil M. P. van der Aalst and Kees M. van Hee. *Workflow Management: Models, Methods, and Systems*. Cooperative Information Systems. MIT Press, Cambridge, MA, USA, 2002. ISBN 978-0-262-01189-1. Cited on pages 1 and 31.

Wil M. P. van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede, and Bartek Kiepuszewski. Advanced workflow patterns. In *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29, Eilat, Israel, September 2000. Springer. Cited on page 1.

Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. Business process management: A survey. In *Proceedings of the 1st International Conference on Business Process Management (BPM 2003)*, number 2678 in Lecture Notes in Computer Science, pages 1–12, Eindhoven, Netherlands, June 2003. Springer. Cited on pages xv, 1, 15, and 16.

Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, et al. The Workflow Patterns initiative, 2012a. URL `http://www.workflowpatterns.com/`. Cited on page 11.

Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, et al. The Workflow Patterns initiative: Evaluations, 2012b. URL `http://www.workflowpatterns.com/evaluations/`. Cited on page 12.

Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, et al. The Workflow Patterns initiative: Vendors, 2012c. URL `http://www.workflowpatterns.com/vendors/`. Cited on page 12.

Wil M.P. van der Aalst, Kees M. van Hee, Arthur H.M. ter Hofstede, N. Sidorova, Eric Verbeek, Marc Voorhoeve, and Moe Thandar Wynn. Soundness of workflow nets: Classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, May 2011. ISSN 0934-5043. Cited on pages 5, 101, 138, 148, and 197.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. Cited on pages 3, 4, 61, 152, 165, and 198.

Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and more focused control-flow analysis for business process models through SESE decomposition. In *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC 2007)*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55, Vienna, Austria, September 2007. Springer. Cited on page 69.

Martin Vasko and Schahram Dustdar. An analysis of web services workflow patterns in Collaxa. In *Proceedings of the 2nd European Conference on Web Services (ECOWS 2004)*, volume 3250 of *Lecture Notes in Computer Science*, pages 1–14, Växjö, Sweden, September 2004. Springer. Cited on page 12.

Bart Verheecke, Wim Vanderperren, and Viviane Jonckers. Unraveling crosscutting concerns in web services middleware. *IEEE Software*, 23(1):42–50, 2006. Cited on pages 2, 31, and 35.

Stephen A. White et al. Business Process Modeling Notation, version 1.0, May 2004. URL `http://www.bpmn.org/`. Cited on pages 1 and 20.

Workflow Management Coalition. Workflow Management Coalition terminology and glossary. Document Number WFMC-TC-1011, Workflow Management Coalition, Winchester, United Kingdom, February 1999. URL `http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf`. Cited on pages xv, 1, 10, 12, 13, and 14.

Bart Wydaeghe. *PacoSuite: Component Composition Based on Composition Patterns and Usage Scenarios*. PhD thesis, Vrije Universiteit Brussel, System and Software Engineering Lab, Brussels, Belgium, November 2001. Cited on page 53.

YAWL Foundation. YAWL Editor, version 2.2.01, September 2011. URL `http://www.yawlfoundation.org/`. Cited on page 22.

Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997. Cited on page 53.