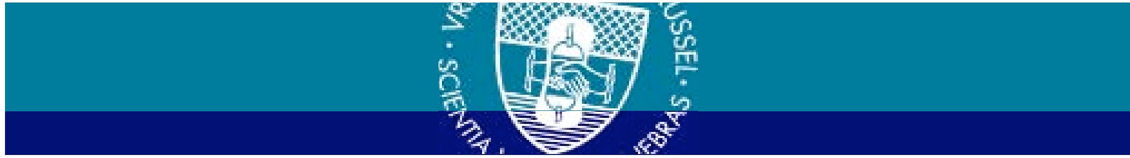


Vrije Universiteit Brussel
Faculteit Wetenschappen
Vakgroep Informatica
System and Software Engineering Lab

Hybrid Aspects for Integrating Rule-Based Knowledge and Object-Oriented Functionality

Maja D'Hondt

May 2004

Jury:

Prof. Dr. Viviane Jonckers, ***advisor***
Vrije Universiteit Brussel (Belgium)
Prof. Dr. Mehmet Akşit
Universiteit Twente (The Netherlands)
Prof. Dr. Olga De Troyer
Vrije Universiteit Brussel (Belgium)
Prof. Dr. Mira Mezini
Technische Universität Darmstadt (Germany)
Prof. Dr. Luc Steels
Vrije Universiteit Brussel (Belgium)

Acknowledgements

First and foremost, I thank my advisor, Viviane Jonckers, for taking me on when I wanted to do a PhD and supporting me these past years. I've learned a lot from her critical mind and constructive approach. Furthermore, I'd like to express my appreciation to the people I have worked with during the course of this dissertation: María Agustina Cibrán, Wim Vanderperren and Davy Suvée for the experiments with existing aspect-oriented approaches, and Kris Gybels for the work on linguistic symbiosis. Also thanks to Agus again and Ragnhild Vanderstraeten for proofreading parts of this dissertation.

My wonderful colleagues and friends have always kept me motivated and provided me with the necessary diversions, so many thanks to Miro Casanova, Ragnhild, Wolfgang De Meuter, Agus, and Bart Verheecke.

My sister Ellie has earned my undying gratitude for helping me with everything from proofreading, fighting with LaTeX, cooking me dinner and implementing the scheduler of the case study in Prolog.

I realise how privileged I am to have my parents, Theo D'Hondt and Raymonda Verstraeten, who support me but also actually understand what I'm talking about. Thanks to Joris Verhaeghe who had to put up with me and my moods every day but managed to keep his patience and my spirits up.

Finally, I'm grateful to Bach, Orbital and Piazzolla for keeping me company during those long long hours in front of my computer screen.

This work has been funded by the *Institute for the Promotion of Innovation through Science and Technology in Flanders (IWTVlaanderen).*

Contents

Table of Contents **i**

List of Figures **vii**

List of Tables **ix**

List of Code Fragments **xiii**

1 Introduction **1**

1.1 Problem Statement . . . 1
1.2 Research Objectives . . . 3
1.3 Aspect-Oriented Programming . . . 4
1.4 Integrating Object-Oriented Functionality and Rule-Based Knowledge . . . 6
1.4.1 Program Integration Issues . . . 6
1.4.2 Aspect-Oriented Programming for Addressing the Program Integration Issues . . . 7
1.5 Hybrid Systems . . . 7
1.5.1 Integrating Object-Oriented and Rule-Based Languages . . . 7
1.5.2 Language Integration Issues . . . 9
1.6 Criteria . . . 10
1.7 Hybrid Composition and Hybrid Aspects . . . 11
1.8 Chapter Summaries . . . 13

2 Rule Objects **17**

2.1 Running Example: E-Commerce . . . 17
2.1.1 Basic Functionality . . . 17
2.1.2 Rule-Based Knowledge . . . 18
Price Personalisation . . . 18
Recommendations . . . 19
2.2 Rule Object Pattern . . . 20
2.2.1 Simple Rule Object . . . 20
2.2.2 Implementing Rule Objects . . . 20
2.3 Integration Patterns . . . 21
2.4 Program Integration Issues . . . 23
2.4.1 Dependency . . . 24
Data Coupling . . . 25
Control Coupling . . . 25
Content Coupling . . . 26
2.4.2 Adaptation . . . 26
2.4.3 Combining Rules . . . 26

2.5 Summary . . . 26

3 Aspect-Oriented Programming 27
3.1 Separation of Crosscutting Concerns . . . 27
3.2 Obliviousness and Quantification . . . 28
3.3 Selecting Aspect-Oriented Approaches . . . 29
3.4 HyperJ for Integrating Rule Objects . . . 31
3.4.1 Dependency . . . 31
3.5 AspectJ for Integrating Rule Objects . . . 33
3.5.1 Dependency . . . 34
Data Coupling . . . 34
Control Coupling . . . 35
Content Coupling . . . 35
3.5.2 Adaptation . . . 36
3.6 JAsCo for Integrating Rule Objects . . . 37
3.6.1 Dependency . . . 37
3.6.2 Combining Rules . . . 38
3.6.3 Dynamic Reconfiguration of Rules . . . 41
3.7 Discussion . . . 41

4 Rule-Based Languages 43
4.1 Overview . . . 43
4.2 Rule Chaining . . . 44
4.2.1 Backward Chaining . . . 45
4.2.2 Forward Chaining . . . 45
4.3 Logic-Based Systems . . . 46
4.3.1 First-Order Predicate Logic . . . 46
4.3.2 Logic Programming . . . 47
4.3.3 Prolog . . . 49
4.4 Production Systems . . . 50
4.4.1 Production Rules . . . 50
4.4.2 Working Memory . . . 51
4.4.3 Conflict Resolution . . . 51
4.4.4 OPS5 . . . 52
4.5 Discussion . . . 53

5 Hybrid Systems 55
5.1 Ruled-Out Systems . . . 55
5.1.1 Frame-Based Languages . . . 56
5.1.2 Rule-Based Knowledge Attached to Frames or Objects . . . 56
5.1.3 Object-Oriented Extensions . . . 57
5.1.4 Translators . . . 57
5.1.5 One-Way Hybrid Systems . . . 57
5.1.6 Rule Management Tools . . . 58
5.1.7 Data-Change-Oriented Systems . . . 58
5.2 Language Integration Issues . . . 58
5.2.1 Example . . . 58
5.2.2 Activating Rules in Objects . . . 60
5.2.3 Returning Inference Results . . . 61
5.2.4 Objects as Rule Values . . . 61

5.2.5 Sending Messages in Rules 61
5.2.6 Returning Message Send Results 61
5.2.7 Rule Values in Methods 62
5.3 Survey of Hybrid Systems 62
5.3.1 Implementation Approaches 62
Symbiotic Systems 62
Symmetric Systems 62
Library Systems 63
5.3.2 Activating Rules in Objects 63
Activation via API 63
Activation in Rule-Based Environments 64
Transparent Activation 64
5.3.3 Returning Inference Results 65
Implicit Result 65
Deterministic Result 65
Collection Result 65
Streamed Result 65
5.3.4 Objects as Rule Values 66
Literals as Rule Values 66
Deconstructed Objects as Rule Values 66
Wrapped Objects as Rule Values 67
Objects as Rule Values 67
5.3.5 Sending Messages in Rules 67
Message Send Statements in Rules 67
Predicate API for Sending Messages in Rules 68
Activation in Object-Oriented Environment 69
Mapping Predicates in Rules onto Methods 69
5.3.6 Returning Message Send Results 69
Explicit Message Send Results 69
Implicit Message Send Results 69
5.3.7 Rule Values in Methods 70
Rule Values as Strings 70
Rule Values as Term Objects 70
Rule Values are Objects 71
Rule Variables as Parametric Types 71
5.4 Discussion 72

6 Model of Hybrid Composition and Hybrid Aspects 75
6.1 Problem and Solution Domains 75
6.2 Criteria 77
6.2.1 Symmetric Obliviousness at the Program Level 77
6.2.2 Symmetric Obliviousness at the Language Level 77
6.2.3 Default and Customised Integration 78
6.2.4 Encapsulated Integration Code 78
6.2.5 Obsolete Issue: Combining Rules 79
6.3 Composition and Aspects are Complementary 79
6.3.1 Elements of the Base Input Programs 80
6.3.2 Characteristics of Composition 80
6.3.3 Characteristics of Aspects 81

6.3.4 Differences between Composition and Aspects 81
6.4 Dynamic vs Static Join Point Models 81
6.4.1 Static vs Dynamic Join Point Models 82
6.4.2 Suitable Model for Hybrid Systems 82
6.5 Base Languages 82
6.5.1 Object-Oriented Base Languages 82
Object-Oriented Behaviour Activation 83
Object-Oriented Join Points 83
Object-Oriented Qualifiers 83
6.5.2 Rule-Based Base Languages 84
Traditional vs New Generation Production Rule Languages . . . 84
Rule-Based Behaviour Activation 85
Rule-Based Join Points 86
Rule-Based Qualifiers 86
6.6 Hybrid Composition 87
6.6.1 General Approach 87
Correspondence between Join Points and Behaviour Activation . 87
Matching between Join Points and Behaviour Activation 89
Alternative Composition Strategies 89
Constraining Hybrid Composition 89
Diagram and Naming Conventions 89
6.6.2 Linguistic Symbiosis with Logic-Based Languages 90
Activating Backward-Chaining Logic Rules from Objects 91
Activating Forward-Chaining Logic Rules from Objects 93
Sending Messages from Backward-Chaining Logic Rules 94
Sending Messages from Forward-Chaining Logic Rules 96
6.6.3 Linguistic Symbiosis with Production Rule Languages 96
Activating Backward-Chaining Production Rules from Objects . 97
Activating Forward-Chaining Production Rules from Objects . . 97
Sending Messages from Backward-Chaining Production Rules . . 98
Sending Messages from Forward-Chaining Production Rules . . . 99
6.7 Hybrid Aspects 99
6.7.1 Hybrid Pointcut Definitions 101
6.7.2 Pointcut Designators 102
Object-Oriented Pointcut Designators 103
Rule-Based Pointcut Designators 103
6.7.3 Hybrid Advice 104
6.7.4 Aspin Variables 105
6.7.5 Advice Argument 105
6.7.6 Temporary Variables 106
6.7.7 Advice Statements 106
6.7.8 Rule-Based and Object-Oriented Proceeds 108
6.7.9 Assignment 108
6.7.10 Unification or Match 109
6.7.11 Message Send 109
6.7.12 Query 110
Logic-Based Query 110
Production Rule Query 110
6.7.13 Assert 110

Logic-Based Assert . . . 111
Production Rule Assert . . . 111
6.8 Conclusion . . . 111
6.8.1 Integration Philosophy . . . 111
6.8.2 Rule-Based Formalisms and Chaining Strategies . . . 112

7 Hybrid Composition and Hybrid Aspects in OReA 113
7.1 The Atelier OReA . . . 113
7.1.1 Architecture . . . 113
7.1.2 SOUL . . . 115
SOUL's Prolog . . . 116
Extending SOUL with Logic-Based Forward Chaining . . . 116
Extending SOUL with "New Generation" Production Rules . . . 118
7.1.3 AspectS . . . 119
7.2 Hybrid Composition in OReA . . . 120
7.2.1 Hybrid Composition of Smalltalk and Prolog . . . 120
Activating Backward-Chaining Logic Rules from Objects . . . 120
Sending Messages from Backward-Chaining Logic Rules . . . 122
7.2.2 Hybrid Composition of Smalltalk and Logic Forward Chaining . . . 124
Activating Forward-Chaining Logic Rules from Objects . . . 124
Sending Messages from Forward-Chaining Logic Rules . . . 125
7.2.3 Hybrid Composition of Smalltalk and Production Rules . . . 126
Activating Forward-Chaining Production Rules from Objects . . . 126
Sending Messages from Forward-Chaining Production Rules . . . 127
7.2.4 Constraining Hybrid Composition . . . 129
7.3 Hybrid Aspects in OReA . . . 129
7.3.1 Hybrid Aspects for Smalltalk and a Prolog-like Language . . . 129
Rule-Based Pointcut Designators . . . 129
Advice Argument . . . 130
Query . . . 130
Message Send . . . 133
Unification . . . 134
7.3.2 Hybrid Aspects for Smalltalk and a Forward-Chaining Logic Language 135
Rule-Based Pointcut Designators . . . 135
Assert . . . 135
Message Send . . . 136
Unification . . . 137
7.3.3 Hybrid Aspects for Smalltalk and a Forward-Chaining Production Rule Language . . . 138
Rule-Based Pointcut Designators . . . 138
Assert . . . 138
Message Send . . . 139
Match . . . 139
7.4 Summary . . . 140

8 Evaluation 141
8.1 Case Study . . . 141
8.1.1 Goal . . . 141
8.1.2 Criteria . . . 142
8.1.3 Setup . . . 142

8.1.4 Scheduling in Business Support Applications . . . 143
8.2 Our Scheduler in Prolog . . . 144
8.3 Objects in Rules . . . 146
8.4 Universal and Customised Integration . . . 147
8.5 Symmetric Obliviousness at the Program Level . . . 151
8.6 Symmetric Obliviousness at the Language Level . . . 153
8.6.1 Activating Rules from Objects . . . 154
8.6.2 Sending Messages from Rules . . . 155
8.7 Encapsulated Integration Code . . . 156
8.8 Summary . . . 156

9 Conclusions **159**
9.1 Summary and Contributions . . . 159
9.2 Future Work . . . 163
9.2.1 Alternative Hybrid Composition Strategies . . . 163
9.2.2 Expressive Pointcut Language . . . 164
9.2.3 Reusable Hybrid Aspects . . . 165
9.2.4 Weaving Technologies . . . 165
9.2.5 Alternative Object-Oriented Languages . . . 166
9.2.6 Aspect Mining and Refactoring . . . 166

A References to the Surveyed Hybrid Systems **169**
A.1 Surveyed Systems . . . 169
A.1.1 Symbiotic Systems . . . 169
A.1.2 Symmetric Systems . . . 169
Logic . . . 169
A.1.3 Library Systems . . . 169
Production Rules . . . 169
Logic . . . 170
A.2 Catalogued Systems . . . 171
A.2.1 Object-Oriented Extensions . . . 171
A.2.2 Linear Logic Languages . . . 171
A.2.3 Translators . . . 172
A.2.4 One-Way Hybrid Systems . . . 172
A.2.5 Rule Managment Tools . . . 172
A.2.6 Data-Change-Oriented Systems . . . 172

B Abstract Grammar for Hybrid Aspect Languages **175**
B.1 Common Part of the Abstract Grammar . . . 175
B.2 Specific Part for Logic-Based Languages . . . 176
B.3 Specific Part for Production Rule Languages . . . 177

C Scheduling Application **179**
C.1 Scheduler in SOUL's Prolog . . . 179
C.2 Scheduler in SOUL's Prolog with Objects and Hybrid Composition . . . 181
C.3 Class Diagram of the Scheduler Domains . . . 183
C.4 Hybrid Aspects for Customising and Calling the Scheduler . . . 183

Bibliography **185**

List of Figures

2.1 A class diagram of the basic functionality of an online store. 18
2.2 An abstract and concrete price personalisation Rule Object. 21
2.3 A class diagram of the extension of the basic functionality of the online store for integration with the Christmas discount rule object. 23
2.4 Dependencies between object-oriented functionality and rule-based knowledge. 24

3.1 Classes of the e-commerce application along feature, rule and class dimensions (adapted from [Ossher & Tarr, 2001]). 32

4.1 Backward chaining steps for inferring a discount percentage. 45
4.2 Forward chaining 46

5.1 Example interaction between object-oriented functionality and rule-based knowledge. 59

6.1 Constraints and variabilities imposed by the problem domain on the solution domain. 76
6.2 Comparison between composition-based and aspect-based approaches. . . . 80
6.3 Diagram and naming conventions. 90
6.4 Activating backward-chaining logic rules from objects. 91
6.5 Returning inference results from backward-chaining logic rules. 92
6.6 Activating forward-chaining logic rules from objects. 93
6.7 Sending messages from backward-chaining logic rules. 94
6.8 Returning message send results to logic rules. 95
6.9 Returning message send results to logic rules using =. 95
6.10 Sending messages from forward-chaining logic rules. 96
6.11 Activating backward-chaining production rules from objects. 97
6.12 Activating forward-chaining production rules from objects. 98
6.13 Sending messages from backward-chaining production rules. 98
6.14 Returning message send results to production rules. 99
6.15 Sending messages from forward-chaining production rules. 100

7.1 The architecture of OReA consisting of existing, basic components (Smalltalk, SOUL and AspectS) and our extensions (other rule-based languages, hybrid aspects and hybrid components). 114
7.2 Pseudo code for the forward-chaining algorithm of the logic-based forward chainer in OReA (adapted from [Russel & Norvig, 1995]). 117

8.1 The generic domain used by our scheduler and a few specialisations of specific domains. 147

List of Tables

3.1 An overview of the integration issues and how they are addressed by the discussed aspect-oriented programming approaches. 42

5.1 An overview of the approach of each surveyed hybrid system for each language integration issue. 74

6.1 Unambiguous correspondence used by hybrid composition between rule-based join points and object-oriented behaviour activation. 88
6.2 Unambiguous correspondence used by hybrid composition between object-oriented join points and logic-based behaviour activation. 88
6.3 Unambiguous correspondence used by hybrid composition between object-oriented join points and production rule behaviour activation. 88

7.1 Uniform interface of rule-based and object-oriented contexts. 130

List of Code Fragments

2.1 Implementation of an abstract discount rule object in Java. 22
2.2 Implementation of the Christmas discount rule object in Java which inherits from the abstract discount rule object. 22
2.3 Implementation of the loyal customer discount rule object in Java which inherits from the abstract discount rule object. 22
2.4 Implementation of the loyal customer rule object in Java. 22
3.1 Concern mappings along the feature dimension in HyperJ (adapted from [Ossher & Tarr, 2001]). 32
3.2 Hypermodule `CheckoutPlusDisplay` in HyperJ (adapted from [Ossher & Tarr, 2001]). 33
3.3 Hypermodule `CheckoutPlusDiscount` in HyperJ (adapted from [Ossher & Tarr, 2001]). 33
3.4 The aspect `EventPricePersonalisation` which describes the event at which discount rules are activated. 34
3.5 The aspect `ApplyXmasDiscount` which activates `XmasDiscountRule`. . . . 35
3.6 An alternative implementation of the `EventPricePersonalisation` aspect which describes the *dynamic* event at which discount rules are activated. . . 35
3.7 The aspect `ApplyLoyalDiscount` which activates `LoyalDiscountRule`. . . . 36
3.8 The aspect `CaptureCustomer` which captures the customer when he or she is checking out and exposes it. 36
3.9 An alternative implementation of the aspect `ApplyXmasDiscount` which activates `XmasDiscountRule` and *adapts* its result. 36
3.10 The aspect bean `XmasDiscountRule` which defines the rule and the hook `XmasDiscountHook`. 38
3.11 The connector `XmasDiscountDeployment` which instantiates the hook of the `XmasDiscountRule` aspect bean. 39
3.12 The connector `XmasAndLoyalDiscountDeployment` which instantiates the hooks of the `XmasDiscountRule` and `LoyalDiscountRules` aspect beans. . 39
3.13 The `CombinationStrategy` interface. 39
3.14 A combination strategy which excludes the execution of `hook B` whenever `hook A` is encountered. 40
3.15 An alternative connector `XmasAndLoyalDiscountDeployment` which uses an exclude combination strategy. 40
7.1 The method `addToShoppingCart:` defined in the class `Customer`. 120
7.2 Rules that conclude the predicate `canBuy:`. 121
7.3 The method `calculatePrice:` in the class `Order`. 121
7.4 Rules that conclude the predicate `discount`. 122
7.5 A rule that concludes `isLocatedInEurope`. 122
7.6 The method `locationIs:` in the class `Customer`. 122

7.7 A rule that concludes the predicate `premierCustomer`. 123
7.8 The method `requestMultimediaComputer` in the class `Store`. 124
7.9 Rules related to concluding the predicate `multimediaComputer`. 124
7.10 Rules that conclude the predicate `addProduct`. 125
7.11 A rule that concludes the predicate `recommend:`. 126
7.12 The method `recommend:` in the class `Customer`. 126
7.13 The method `createCustomerNamed:withChargeCard:` in the class `Store`. . 127
7.14 A production rule that concludes the status of a customer. 128
7.15 The method `hasChargeCard:` in the class `Customer`. 128
7.16 A production rule that concludes gift wrappers to be added to a shopping cart. 128
7.17 The method `addToShoppingCart:` defined in the class `Customer`. 130
7.18 Rules that conclude the predicate `canBuy:`. 131
7.19 The hybrid aspect that integrates the method `addToShoppingCart:` defined in `Customer` and rules that conclude the predicate `canBuy:`. 131
7.20 The method `calculatePrice:` in the class `Order`. 132
7.21 Rules that conclude the predicate `discount`. 132
7.22 The hybrid aspect that integrates the method `calculatePrice` defined in `Order` and rules that conclude the predicate `discount:`. 132
7.23 The method `isLocatedIn:` defined in the class `Customer`. 133
7.24 The hybrid aspect that integrates rules that conclude `isLocatedInEurope:` and the method `isLocatedIn:` defined in the class `Customer`. 133
7.25 A rule that concludes the predicate `hasMultimediaScreen`. 134
7.26 The method `getCompatibleScreens` in the class `Computer`. 134
7.27 The hybrid aspect that integrates rules that conclude `hasMultimediaScreen` and the method `getCompatibleScreens:` defined in `Computer`. 135
7.28 The method `addProduct:` in `ShoppingCart`. 135
7.29 Rules that conclude the predicate `inCart:`. 136
7.30 The hybrid aspect for integrating the method `addProduct:` in `ShoppingCart` with rules that conclude the predicate `inCart:`. 136
7.31 The hybrid aspect for integrating rules that conclude the predicate `inCart:` with the method `addProduct:` defined in `ShoppingCart`. 136
7.32 The method `createOrder:` defined in the class `Store`. 137
7.33 The hybrid aspect that integrates the method `createOrder:` defined in `Store` with rules that conclude the predicate `inCart:`. 138
7.34 The hybrid aspect for integrating the method `createOrder:` defined in `Store` with production rules that conclude products to be added to a shopping cart. 139
7.35 The hybrid aspect for integrating an alternative wrapper with a production rule that adds gift wrappers for products to a shopping cart. 139
7.36 The hybrid aspect with delete operator for integrating the method `createOrder:` defined in `Store` with production rules that conclude products to be added to a shopping cart. 139
7.37 The hybrid aspect for integrating the method `calculatePrice` defined in `Order` with asserting products. 140
8.1 The top-level scheduling rule that concludes `canMoveTo:and:`, implemented in standard Prolog. 145
8.2 The rule that concludes `noOverlap` and the rule that concludes `overlap`, implemented in standard Prolog. 145
8.3 The rules that conclude `overlap:` and the auxiliary rule that concludes `overlapAux:`, implemented in standard Prolog. 146

8.4 A would-be top-level scheduling rule that concludes `canMoveTo:and:in:` in SOUL with objects. 148
8.5 Would-be rules that conclude `noOverlapIn:` and `overlapIn:` in SOUL with objects. 148
8.6 The auxiliary rule that concludes `overlapAux:`, implemented in SOUL with objects. 148
8.7 The method `overlapAux:` defined in the class `Slot`. 149
8.8 The real top-level rule that concludes `canMoveTo:and:`, implemented in SOUL with objects and hybrid composition. 149
8.9 The method `moveTo:in:` defined in the class `ScheduledItem`. 150
8.10 The rule that concludes `noOverlapIn:` and the rule that concludes `overlapIn:`, implemented in SOUL with objects and hybrid composition. 150
8.11 The rules concluding the predicates `canMoveTo:and:in:`, `noOverlapIn:with:` and `overlapIn:with:` after manual and invasive customisation for scheduling courses in schools. 151
8.12 The hybrid aspect `CaptureItem`. 152
8.13 The hybrid aspect `SchoolOverlap`. 152
8.14 The hybrid aspect `TVOverlap`. 153
8.15 An alternative method `moveTo:in:` defined in `ScheduledItem`. 154
8.16 A hybrid aspect that checks if an item can be moved. 154
8.17 A hybrid aspect for inferring available slots for a newly created course. . . 155
8.18 A hybrid aspect for retrieving the initial slots transparently. 156

Chapter 1

Introduction

1.1 Problem Statement

The complexity of software applications is steadily increasing and knowledge management of businesses is becoming more important. The real-world domains of many software applications, such as e-commerce, the financial industry, television and radio broadcasting, hospital management and rental business, are inherently knowledge-intensive. Current software engineering practices result in software applications containing implicit knowledge about the domain or the business that is *tangled* with the *core functionality*. Currently, the state of the art in software engineering uses object-oriented programming languages for implementing the core functionality. Knowledge comes in many forms [Schreiber et al., 2000], such as domain knowledge, consisting of concepts, relations and constraints, and *rule-based knowledge*, which states how to infer new concepts and relations. *Rules* typically represent knowledge about policies, preferences, decisions, recommendations and advice, to name but a few. In this dissertation, we focus on rule-based knowledge in object-oriented software applications. Nowadays, the popularity of *business rules* [Ross, 2003; von Halle, 2001; Date, 2000] has caused a revival of the notion of explicit rule-based knowledge, now in the context of object-oriented software applications. However, business rules represent but a part of rule-based knowledge. Knowledge-intensive subtasks of software applications, such as (semi-)automatic scheduling, intelligent help desks and advanced support for configuring products and services, are also instances of rule-based knowledge that are typically more sophisticated than what is understood by business rules.

Real-world domains are subject to change and businesses have to cope with these changes in order to stay competitive. Additionally, a family of applications may have the same core functionality but varying rule-based knowledge. Therefore, it should be possible to identify and locate the software's rule-based knowledge easily and adapt it accordingly while at the same time avoiding propagation of the adaptations to the core functionality. Similarly, when the core functionality is subject to change, we should be able to update or replace it in a controlled and well-localised way.

Furthermore, the development of software where rule-based knowledge and core functionality are tangled is a very complex task: the software developer, who is typically a software engineer but not a knowledge engineer, has to concentrate on two facets of the software at the same time and manually integrate them. This violates the principle of *separation of concerns* [Dijkstra, 1976; Parnas, 1972; Hürsch & Lopes, 1995], which states that the core functionality should be separated from other concerns or aspects, such as in this case rule-based knowledge. In short, the tangling of rule-based knowledge and core

functionality makes understanding, maintaining, adapting, reusing and evolving the software difficult, time-consuming, error-prone, and therefore expensive. Although separation of concerns holds for the entire software development cycle, we focus on the implementation level in this dissertation.

When considering literature on developing software applications with rule-based knowledge we find that all advocate making rules explicit and separating them from the object-oriented core functionality [von Halle, 2001; Ross, 2003; Date, 2000]. Moreover, many technologies exist that are targeted towards these goals, although they take radically different approaches. First of all, rule-based knowledge can be represented separately in the object-oriented programming language itself. An extension to this approach is representing rule-based knowledge explicitly using object-oriented design patterns, referred to as the *Rule Object Pattern* [Arsanjani, 2001]. Other approaches focus on externalising explicit rules, such as *Business Rule Beans*, which store rules as XML fragments [Rouvellou et al., 2000]. Finally there are dozens of both commercial and academic *hybrid systems*, which integrate a full-fledged rule-based language with a state-of-the-art object-oriented programming language.

This abundant choice of approaches notwithstanding, we observe that the actual tangling of rule-based knowledge and object-oriented functionality is not resolved by any of them. Although the rule definitions — in whichever format or language — are physically separated from the object-oriented program, the code that integrates them is still tangled in the programs themselves. This makes the rule-based knowledge and object-oriented functionality dependent on each other. This in turn results in change propagation, which impedes maintenance, reuse and evolution.

In this dissertation we focus on the aforementioned hybrid systems for representing rule-based knowledge (in the rule-based language) and object-oriented functionality (in the object-oriented language). Rule-based languages provide dedicated language constructs for representing rules and manage the flow of rules automatically. This gives them a considerable advantage over the other approaches. An additional challenge introduced by this development context is that two languages of different programming paradigms are integrated. Our survey of existing hybrid systems shows that there are plenty of paradigm leaks[1] from the one language in the other: the languages either adopt features of one another or provide explicit integration interfaces. Unfortunately, these paradigm leaks even strengthen the dependencies between rule-based knowledge and object-oriented functionality. Moreover, the software engineer has to become familiar with the technologies or languages used by the knowledge engineer and vice versa.

Note that we do not consider *knowledge-intensive applications* that are entirely developed using knowledge engineering [Schreiber et al., 2000], since our starting point remains object-oriented software engineering. The advantages of combining the rule-based and object-oriented paradigms have been recognised by many, which also explains the existence of so many hybrid systems. Furthermore, information systems are also not taken into account, although some database management systems offer support for business rules. The reason is that they implement a *data-change-oriented* approach, triggering rules when data changes. However, when rules are not bound to a particular object or data but are "free-floating", a *service-oriented* approach is warranted [von Halle, 2001]. Moreover, even C. J. Date states

[1] This term is coined in [Brichau et al., 2002].

that not all rules can be implemented in the database layer, but have to be considered in the application layer [Date, 2000].

1.2 Research Objectives

The multitude and popularity of technologies and methodologies for developing object-oriented software applications with explicit rule-based knowledge indicates that addressing the aforementioned problems would be a useful endeavour.

The first problem, dependencies between rule-based knowledge and object-oriented functionality, is addressed by *encapsulating* the integration code that is tangled in the separated programs. Then, the dependencies have to be *inverted*: instead of the rule-based knowledge and object-oriented functionality being dependent on each other, the encapsulated integration depends on them. That way, both rule-based knowledge and object-oriented functionality are *oblivious* to being integrated with one another. This enables the development of alternative integrations when either one of them changes.

Note that when either rule-based knowledge or object-oriented functionality are not meaningful in their own right, but are intended to be integrated with other functionality, obliviousness is still crucial. Indeed, obliviousness reduces coupling between programs. This means that each program can evolve independently and can be reused in other integrations without it having to carry around the extra baggage of all the integrations it is part of.

The second problem, paradigm leaks between the rule-based and object-oriented languages in a hybrid system, is addressed by making any connection between the two languages transparent. We assert that our approach for addressing the first problem enables a solution for the second: when the encapsulated integration code is expressed in a hybrid language which amalgamates features of the two languages being integrated, it serves as a paradigmatic buffer. As such, both languages in a hybrid system remain oblivious to one another. Whereas the former obliviousness is at the *program* level, this one resides at the *language* level.

Aspect-Oriented Programming is a principle that identifies the need for alternative modularisations for encapsulating tangled code, referred to as *aspects* [Kiczales et al., 1997]. These modularisations complement the standard abstraction mechanisms of a programming language, whether they are procedures, functions, objects or any other. One of the main characteristics of aspect-oriented programming approaches is support for *obliviousness*. This leads us to our research hypothesis:

> Aspect-oriented programming enables encapsulating the integration between rule-based knowledge and object-oriented functionality in hybrid systems while maintaining obliviousness both at the program level and the language level

In the remainder, we introduce aspect-oriented programming, the issues related to integrating rule-based knowledge and object-oriented functionality, hybrid systems and their language integration issues, the criteria which potential solutions must meet, and finally our approach of hybrid composition and hybrid aspects. The chapters of this dissertation are summarised at the end.

1.3 Aspect-Oriented Programming

A software application involves many and heterogeneous concerns. By concerns we refer to properties or areas of interest in the system. Typically, concerns can range from high-level notions like security and quality of service to low-level notions such as caching and buffering. To deal with all these heterogeneous concerns in a software application *separation of concerns* is fundamental: it refers to the ability to identify, encapsulate and manipulate only those parts of the software that are relevant to a particular concept, goal or purpose [Parnas, 1972].

Generally, programming languages provide a dominant decomposition mechanism, such as procedures, functions or objects, for encapsulating concerns. However, this dominant decomposition mechanism is not always suitable for capturing and representing all kinds of concerns that can be found in a software application. Typically, there exist "system-wide" concerns that cut across the natural modules of the application and cannot be encapsulated by the dominant decomposition mechanism of a language [Kiczales et al., 1997]. This problem is referred to as *the tyranny of the dominant decomposition* [Tarr et al., 1999]. Well-known examples of *crosscutting concerns* in object-oriented programs are systemic properties or behaviours such as logging, error recovery and synchronisation.

Aspect-oriented programming focuses on mechanisms for achieving a better separation of crosscutting concerns in software applications at the implementation level. Consider a program implemented in a standard procedural or object-oriented programming language. We refer to such a program as a *base program* and the language as a *base language*. A crosscutting concern typically affects this base program in many, non-local places. Aspect-oriented programming approaches provide mechanisms for expressing such crosscutting concerns in an encapsulated way and separately from the program they affect. Most approaches but not all provide a new kind of abstraction for representing crosscutting code in an encapsulated way.

Many adopt the terminology of *AspectJ* [Kiczales et al., 2001], probably because it is one of the first aspect-oriented approaches and one of the best known. Additionally, many newer approaches are based on the AspectJ approach, for the same reasons. However, there are approaches that provide very different support for encapsulating crosscutting concerns.

According to AspectJ terminology, the places that are affected by a crosscutting concern are referred to as *join points*. These can be dynamic — points in the execution of a program — or static — elements of a program definition. AspectJ-like approaches provide a new kind of abstraction, an *aspect*, for encapsulating a crosscutting concern. Typically, the aspect represented in a general-purpose aspect language. Integrating the base program with an aspect program at the specified join points is a process called *weaving*. Conceptually, the result is that the base program itself or its execution is weaved at each specified join point with a means for effecting semantics [Masuhara & Kiczales, 2003]. The means for effecting semantics are defined in *advice*. Join points are specified using a *pointcut*. As such, an aspect consists of pointcut and advice definitions.

We use this terminology throughout the dissertation, unless it is not compatible with a particular aspect-oriented approach being discussed.

Not all approaches employ a separate aspect language for expressing crosscutting concerns. The modelling framework presented in [Masuhara & Kiczales, 2003] attempts to unify different kinds of aspect-oriented approaches and provides concrete models for some existing approaches, instantiated from the framework. We argue in the dissertation that two kinds of approaches are suitable for integrating rule-based knowledge and object-oriented

functionality in hybrid systems. The first is based on *composition* and the second on the aforementioned *aspects*.

A composition-based approach takes two input programs and composes them according to a composition rule. This single composition rule results in multiple mappings between automatically identified join points on the one hand, and means for effecting semantics on the other. As such, composition provides an automated and general composition strategy.

Conversely, an approach based on pointcuts and advice, specifies several heterogeneous join points at which the advice will be weaved. However, this typically does not constitute a general integration strategy. Nevertheless, a major advantage of aspects, is that they are able to encapsulate any integration-dependent code which might be needed for adapting the base programs to one another.

In our opinion, approaches based on composition and aspects complement each other. Intuitively, we can say that composition allows one to define an overall default strategy for composing two programs automatically, based on a mapping between program or execution elements. Aspects, on the other hand, allow very selective and customised integration of programs, by specifying a set of heterogeneous join points and defining advice that is to be weaved at these join points. In order to combine both approaches, a common join point model is required.

Fillman argues that the distinguishing characteristics of aspect-oriented programming approaches are *quantification* and *obliviousness* [Filman, 2001]:

- Quantification relates to support for specifying many, non-local and heterogeneous join points. Many aspect-oriented systems support *dynamic* quantification, which weaves input programs depending on run-time properties [Filman & Friedman, 2000].

- Obliviousness, on the other hand, is the idea that a program does not have to anticipate its being weaved or composed with other programs. Hence, the program does not have to foresee and declare the join points explicitly. As a result, a base program does not depend on an aspect it is weaved with or another base program it is composed with. When aspects are used, they depend on the base program instead of the other way around [Nordberg III, 2001]. In the case of composition-based approach, the base programs are both oblivious to being composed.

Although the primary and best-known goal of aspect-oriented programming is modularising crosscutting concerns — hence the need for quantification — it is also a very powerful means for achieving minimal dependencies between modules or classes. This is mainly due to the other characteristic of aspect-oriented approaches: obliviousness. This is accomplished with dependency inversion, so that the aspect or composition itself is dependent on the programs it weaves or composes rather than the programs being dependent on each other. A program that is oblivious to being weaved with another program can be reused in other integrations, without it having to carry around the extra baggage of all the integrations it is part of.

Both [Hannemann & Kiczales, 2002] and [Nordberg III, 2001] show the advantages of dependency inversion with aspect-oriented programming in the context of design patterns.

1.4 Integrating Object-Oriented Functionality and Rule-Based Knowledge

We identify a number of issues that occur when integrating object-oriented functionality and rule-based knowledge. These *program integration issues* are independent of the representation of rule-based knowledge, whether it is expressed as an object-oriented program, externalised, or implemented in a rule-based language. We also discuss these issues in [D'Hondt et al., 1999], [Cibrán et al., 2003a], [Cibrán et al., 2003b] and [D'Hondt et al., 2004].

When representing rule-based knowledge as rule objects — object-oriented design patterns — it is possible to experiment with existing aspect-oriented approaches and investigate their ability to address the identified integration issues. These results are also presented in [Cibrán et al., 2003a] and [Cibrán et al., 2003b].

1.4.1 Program Integration Issues

- **Dependency**
  An age-old principle for avoiding change propagation and improving reuse is *loose coupling* of software parts. This principle is first introduced in [Stevens et al., 1974] and extended for object-oriented software development in [Eder et al., 1992]. Coupling is a measure of interconnection among software parts. It depends on the interface complexity between these parts, the point at which entry or reference is made to a part, and what data passes across the interface. Interesting to note is that another goal of minimal coupling is to allow teams of developers to work independently on their modules [Harrison & Coplien, 1996]. We identify three forms of coupling when integrating object-oriented functionality and rule-based knowledge, of which two are examples of *tight* coupling.

  Object-oriented functionality and rule-based knowledge depend directly on each other: at certain events in the former, the latter is activated and vice versa. In both directions, values are passed and returned, which is *data coupling*, a low level of coupling. However, most approaches for separating rule-based knowledge allow it to access instance variables of the object-oriented functionality directly, which is the worst form of coupling: *content coupling*. Furthermore, we identify another form of tight coupling, *control coupling*, when the activation of rule-based knowledge depends on the run-time properties of the object-oriented functionality.

  Even low coupling such as *data coupling*, which consists of communication via parameters, complicates evolution. When the rule-based knowledge evolves or other rules have to replace it, other inputs are possibly required from the object-oriented functionality. Hence, the change in rule-based knowledge propagates to the object-oriented functionality.

- **Adaptation**
  When integrating two programs, incompatibilities between them due to evolution necessitate adaptations. Hence, integration consists of more than a simple call from rule-based knowledge to activate object-oriented functionality or the other way around. The programs are extended with some kind of "glue code" for converting or manipulating the results of behaviour activation.

  Usually, approaches for separating rule-based knowledge require the adaptations to be expressed as extensions to objects. Hence, the object-oriented functionality is

cluttered with code to accommodate each of its integrations with different rule-based knowledge.

- **Combining Rules**
  When the approach for representing rule-based knowledge does not deal with combinations of rules, more code needs to be written for expressing this explicitly and manually.

1.4.2 Aspect-Oriented Programming for Addressing the Program Integration Issues

Most existing aspect-oriented approaches use an object-oriented language as the base language. Therefore, when expressing both the core functionality and the rule-based knowledge as object-oriented programs, existing aspect-oriented approaches can be used for addressing the aforementioned integration issues. Earlier, we identified two different kinds of aspect-oriented approaches that are useful for integrating rule-based knowledge and object-oriented functionality. Therefore, a valuable experiment is to investigate how representatives of each kind deal with the program integration issues.

First of all, [Ossher & Tarr, 2001] indicates how business rules can be separated using *HyperJ*, an example of a composition-based approach. Furthermore, we show in [Cibrán et al., 2003a] and [Cibrán et al., 2003b] how *AspectJ* and *JAsCo*, respectively, deal with the integration issues. Both are representatives of a pointcut/advice approach to aspect-oriented programming, the main difference being that JAsCo endeavours to make aspect less dependent on base program with which they are weaved. The work on HyperJ and both our experiments with the other approaches show that aspect-oriented programming is indeed able to address the program integration issues in a purely object-oriented development context. However, this dissertation wishes to consider hybrid systems as development context, where rule-based knowledge and object-oriented functionality are both represented in their most suitable language, i.e. rule-based and object-oriented languages, respectively. Therefore, the application of aspect-oriented programming becomes less straightforward.

1.5 Hybrid Systems

This section introduces hybrid systems and presents the issues related to integrating the object-oriented and rule-based languages. We present the language integration issues in [D'Hondt et al., 2004] as well.

1.5.1 Integrating Object-Oriented and Rule-Based Languages

The object-oriented paradigm became increasingly successful both in the field of software engineering and in the field of artificial intelligence. These fields emphasise a different point of view about the notion of an object, respectively [Masini et al., 1991]:

- from a structural point of view, an object defines a data structure and a set of operations applicable to this structure

- from a conceptual point of view, an object represents a knowledge concept

Usually, object-oriented languages for software engineering are class-based. The historical leaders of these family of languages are *Simula* and *Smalltalk-80*. The latter is often considered as the archetype of class-based object-oriented programming languages because

of its uniform model: the only data entity is the object and the only control structure is message sending.

Object-oriented languages in the field of artificial intelligence on the other hand, are usually referred to as *frame-based languages*. They are derived from network-based formalisms and Minsky's frame idea [Minsky, 1975]. They are usually prototype-based, where a frame describes a standard or typical situation or object. A frame holds a number of slots, each described by a number of facets, the most trivial being the slot's value. Contrary to objects from software engineering, a frame has no behaviour described by methods but local operations held by the slots which are activated on slot access. Typically, these are procedures written in the underlying programming language, such as LISP as in, for example, *KL-ONE* and *KRL*.

Nowadays, popular object-oriented programming languages for software engineering are class-based languages with sophisticated development environments and extensive libraries, in particular *C++*, *Java* and *Smalltalk*.

Rule-based reasoning emulates human thought and problem solving based on the explicit representation of human knowledge. We define *rule-based systems* as systems or languages that provide

1. *rules*, modular structures for expressing this knowledge,
2. data structures that are manipulated by active rules, and
3. an inference engine or interpreter for selecting and activating rules.

Note that this a restricted version of the definition for *pattern-directed inference systems* [Jackson, 1986].

Rules are conclusion–condition pairs. There exist two main formalisms for expressing rules: *production rules* and *first-order predicate logic* [Jackson, 1986]. Systems based on the former formalism are usually referred to as *production systems* and express knowledge in *production rules*, whereas systems based on the latter are often called *logic-based systems* where rules are *logic clauses*. In this work, we use the terms rule-based system and rule to denote the corresponding items in both formalisms in a general way.

The engine activates a rule when incoming data matches the activation pattern of either its condition or its conclusion. The second approach is taken by *backward-chaining* engines: they attempt to prove a conclusion by finding rules that would conclude it and trying to establish their conditions. *Forward-chaining* engines activate a rule when data matches its condition and generate its conclusion. Backward chaining is goal-oriented and only activates rules that potentially contribute to proving the goal. Forward chaining is data-oriented and generates all possible inferences – even irrelevant ones – but does not require a specific goal. Typically, analytic tasks such as classification, diagnosis and assessment are goal-oriented and solved by backward reasoning, whereas synthetic tasks such as design, scheduling and assignment are usually solved by forward reasoning [Schreiber et al., 2000].

We refer to *hybrid systems* as systems or languages that combine forms of object-oriented programming and rule-based reasoning. The first hybrid systems surface in the field of artificial intelligence and combine frames and rules. They were developed to address the difficulty of representing knowledge that is naturally expressed as rules, in an object-oriented style [Jackson, 1986], and vice versa [Fikes & Kehler, 1985]. These hybrid systems typically provide declarative as well as procedural facets to be associated with a slot in a frame. Declarative facets attach collections of rules to a slot which are activated when the slot is

accessed. *KEE* [Hendrix, 1979], *LOOPS* [Stefik et al., 1983] and *CENTAUR* [Aikins, 1984] are the most notable hybrid systems of this kind.

Currently, there is a new surge of hybrid systems based on state-of-the-art object-oriented programming languages: C++, Java, and Smalltalk. They implement a powerful but efficient rule-based system in one of these languages and use objects as common data structure. We provide a thorough survey of these systems in this dissertation.

The current hybrid systems differ from the first hybrid systems in that rules are not attached to frames (or objects) and activated when frame slots are accessed. Conversely, rules and objects are defined separately.

1.5.2 Language Integration Issues

We identify a number of issues when integrating a rule-based language and an object-oriented language while maintaining language obliviousness. The issues arise when the paradigmatic distance between the languages get in the way of a transparent integration. There are six issues, which can be divided in two groups: one group is concerned with activating rules from an object-oriented program (issues 1 to 3) whereas the other group deals with sending messages from rules (issues 4 to 6).

1. **Activating Rules in Objects**
   Object-oriented languages in their purest form have a single means for activating behaviour, and that is by sending messages to objects. In the context of a hybrid system, an object-oriented language also needs to support the activation of behaviour in the rule-based language.

2. **Returning Inference Results**
   When a message is sent in an object-oriented program, a value which is the result of evaluating the corresponding method is returned, even if it is a null value. Hence, when an object-oriented program activates rules, the result of evaluating them — the inference results — also have to be returned. However, inference results of forward-chaining rules are side-effects and cannot be returned. A backward chainer, however, binds values to rule variables as a result of inferencing. Hence, the approach to "returning" inference results has to be consolidated with the way message send results are returned. Another discrepancy is that rule-based languages manipulate single or multiple inference results in the same way, whereas object-oriented programs expect a single result.

3. **Objects as Rule Values**
   Since in a pure object-oriented language, objects are the only data, objects are passed to the rule engine when it is activated from an object-oriented program. Therefore, the rule engine needs to be able to bind rule variables to objects. Moreover, the pattern matcher or unification algorithm needs to be adapted in order to match or unify objects.

4. **Sending Messages in Rules**
   A rule engine activates behaviour when it attempts to infer a rule's condition in both backward chaining and forward chaining — although achieved differently. Moreover, a forward chainer also activates behaviour when it generates a rule's conclusion. In a hybrid system, the rules need to be able to send messages.

5. **Returning Message Send Results**
   The rule engine expects an indication of failure or success when it attempts to establish

a condition. Hence, when a message is sent in order to establish a condition, it has to be interpreted as returning success or failure. Moreover, message sends might return objects and not just boolean values (which are also objects in pure object-oriented languages). Therefore, the rule-based language has to foresee a mechanism for binding a message send result to a rule variable in order to use it while inferencing. Again, there has to be a way to indicate whether a collection of objects is a single binding or multiple bindings for a certain rule variable.

6. **Rule Values in Methods**
   Rule variables are bound to rule values while inferencing. When a message is sent in a rule, these rule values are passed to the object-oriented language. Moreover, a special kind of rule value is an unbound rule variable. Since an active rule can contain unbound rule variables, these variables can also become parameters in a message send. Additionally, when the object-oriented program wants to activate an unbound query in order to employ the inferencing capabilities fully, unbound rule variables need to be created ex nihilo.

We conduct a survey of existing hybrid systems with respect to the language integration issues and discern two different integration philosophies.

One philosophy is to adapt at least one of the integrated languages to be more compatible with the other. Most modern-day hybrid systems based on production rules follow this approach and extend the production rule language with object-oriented features such as message sending. This integration philosophy allows for a very straightforward integration with the object-oriented language: a message send in a production rule is always evaluated in the object-oriented language. Of course, this philosophy results in paradigm leaks right from the beginning, since the object-oriented paradigm materialises in the production rule language.

This contrasts with the other integration philosophy, the one employed by most logic-based hybrid systems. In such systems, both languages are outfitted with an interface to the other language, expressed in their own language constructs. Special keywords or characters notify the evaluator of a dispatch to the other language. Moreover, both languages usually have some kind of representation of values of the other language which they can access. Consequently, this philosophy also results in paradigm leaks.

We observe that when some degree of language obliviousness in existing hybrid systems is achieved, it is often at the expense of automatic integration. Indeed, transparent integration is usually enabled by a manual specification of how certain program elements of the different languages map. On the other hand, automation typically results in a less flexible integration. This is because automation is enabled by a one-to-one mapping between program elements of the different languages from which cannot be deviated.

1.6 Criteria

From the integration issues at the program level and the language level, we distil a set of criteria that we feel an approach to integrating rule-based knowledge and object-oriented functionality in hybrid systems must meet. Moreover, a potential approach should reflect the variability in the problem domain. More specifically, the approach has to consider both rule-based formalisms and both rule-based chaining strategies. When evaluating our approach, we show how our approach addresses the criteria.

The criteria are:

- **Symmetric Obliviousness at the Program Level**
  The introduction of hybrid systems as a development context makes it clear that obliviousness should be *symmetrical*: both object-oriented functionality and rule-based knowledge should be oblivious to being integrated with each other.

- **Symmetric Obliviousness at the Language Level**
  Existing hybrid systems have several *paradigm leaks* in both object-oriented functionality and rule-based knowledge that signal the presence of another paradigm. Therefore, obliviousness at the program level can only be achieved when there is also obliviousness at the language level: representing a program in a certain language should not reveal the integration with programs in a language of another paradigm. Again, this should be achieved for both languages that are integrated, resulting in symmetric obliviousness at the language level.

- **Default and Customised Integration**
  Integrating a program in one language with a program in another language in a transparent way is often achieved at the expense of automatic integration. On the other hand, automation often results in a less flexible integration. We want the best of both worlds: support for an automatic default integration and support for overriding this integration in a highly customised way.

- **Encapsulated Integration Code**
  Customised integration needs to be able to deal with programs that are incompatible as a result of unanticipated integration. Hence, *integration code*, specific to a certain integration, has to be encapsulated.

Note that when using a rule-based language for representing rules, the responsibility for dealing with the combination of rules shifts from the integration to the rule-based knowledge. Therefore, considering our hybrid systems context, we do not adopt it as a criterion.

1.7 Hybrid Composition and Hybrid Aspects

We propose a combination of composition and aspects for integrating object-oriented functionality and rule-based knowledge in hybrid systems, referred to as *hybrid composition* and *hybrid aspects*. Our approach considers two base languages of a different programming paradigm instead of one single base language. In order to achieve this, we introduce several new concepts and mechanisms. We review these contributions below.

We argue that hybrid systems impose a dynamic join point model: existing hybrid systems typically integrate programs in terms of the execution of the programs rather than merge the program definitions themselves. Indeed, most hybrid systems provide some way of giving control from the one language evaluator to the other. Therefore, we have to identify dynamic join points in rule-based languages, referred to as **rule-based join points**, similar to existing dynamic object-oriented join points. Furthermore, we want to consider join points for rule-based languages with different rule-based formalisms and chaining strategies.

We provide a **combination of composition and weaving**, more specifically we combine hybrid composition and hybrid aspects. In order to achieve this, we need to construct a unified, dynamic join point model that is employed by these two aspect-oriented approaches.

A third contribution is that hybrid composition requires a **linguistic symbiosis**[2] between elements of both input programs, one rule-based and one object-oriented. This consists of a correspondence between join points in the one language and behaviour activation in the other and a one-to-one mapping between corresponding elements. Although this ensures transparent activating of rules in objects and sending messages in rules, it does not deal with the other language integration issues transparently. Both input languages still need to be extended with features of the other in order to address the remaining issues. Yet, it does enable a universal and automatic integration. Linguistic symbiosis is also presented in [D'Hondt et al., 2004].

Hybrid aspects have to define rule-based pointcuts and hybrid advice, which requires the conception of a **hybrid aspect language**, a fourth contribution. This aspect language amalgamates features of both input languages in order to deal with the language integration issues: statements for activating behaviour in both languages, objects and rule values, and operators for "returning" inference and message send results. Furthermore, hybrid advice is parametrised with values from the interrupted execution context, which can be either objects or rule variables. Hence, it has to be able to manipulate both kinds of values. We report on hybrid aspects in [D'Hondt & Jonckers, 2004]. This dissertation provides an abstract grammar of our hybrid aspect language which takes variations with respect to rule-based formalism and chaining strategy into account.

A fifth contribution consists of three proof of concept implementations in our atelier *OReA* of hybrid composition and hybrid aspects:

- for integrating Smalltalk with a Prolog-like language (logic-based and backward-chaining)
- for integrating Smalltalk with a logic-based forward chainer
- for integrating Smalltalk with a language such as the new generation OPS5-like languages (based on production rules and forward-chaining)

This dissertation evaluates our approach against the criteria. In this introduction, we can only indicate how our approach meets the criteria. Both kinds of aspect-oriented approaches enable dependency inversion which results in symmetric obliviousness at the program level. Hybrid composition enables a universal and automatic integration of object-oriented functionality and rule-based knowledge. However, since no separate aspect language is employed, each integrated language still reveals features of the other paradigm. Hybrid aspects, on the other hand, allow customised integration and encapsulate hybrid integration code in an expressive way. Moreover, the aspect language is a hybrid language which absorbs any paradigm leaks and therefore results in obliviousness at the language level. A combination of composition and aspects leads to obliviousness at the language level as well as default and customised integration.

Finally, we would like to point out a more far-reaching consequence of our work in the context of multiparadigm or multilanguage programming environments. We provide an original and valuable approach to achieving obliviousness at the language level in multilanguage programming environments: an intermediate hybrid language is used for expressing the integration of the languages. This hybrid language consists of behaviour activating statements and dispatches behaviour activation to the correct language evaluator. Furthermore, it contains features of the original languages for manipulating and converting values that

[2] This term is first coined in [Steyaert, 1994].

are exchanged. Although our approach requires the design of a new hybrid language which combines features of the integrated languages, it reuses large parts of the actual evaluators of these languages. Also, there is complete backward and forward compatibility between programs written in a particular language, whether operating as a stand-alone language or integrated with another language using alternative hybrid aspects. Finally, as mentioned earlier, our approach maintains language obliviousness, which allows developers to program in their favourite language without having to deal with alien language features. We demonstrate this approach for the combination of rule-based and object-oriented languages with hybrid aspects. Yet we believe that the same general approach can be considered for other language and paradigm combinations.

1.8 Chapter Summaries

Chapter 2: Rule Objects This chapter introduces the problem of state-of-the-art integration of object-oriented functionality and rule-based knowledge, which we refer to as the lack of obliviousness at the program level. In order to illustrate this, we consider an instance of rule-based knowledge which is becoming increasingly popular nowadays: business rules. We show how separated business rules are integrated with the core functionality in an object-oriented software development context using a pattern-based approach. This approach results in what we refer to as program integration issues, i.e. problems achieving obliviousness. These issues concern dependencies between rules and core functionality, which result in change propagation and thus complicate important software engineering tasks such as maintenance, evolution and reuse.

Chapter 3: Aspect-Oriented Programming The next step in this dissertation consists of introducing one of the relevant fields in the familiar object-oriented development context: aspect-oriented programming. One or two exceptions aside, one of the distinguishing characteristics of current aspect-oriented programming approaches is that they support obliviousness when integrating object-oriented programs. Therefore, we take the natural step of applying existing aspect-oriented approaches for integrating rule-based knowledge and object-oriented functionality in an object-oriented development context. As such, we first briefly review the different kinds of aspect-oriented programming approaches and provide a more thorough account of the selected ones: a composition-based and two pointcut/advice-based approaches.

Chapter 4: Rule-Based Languages Since our target development context is hybrid systems, which combine an object-oriented programming language with a rule-based language, we introduce rule-based languages in this chapter. In this dissertation, we use the general term rule-based language to refer to languages of the logic or production rule formalisms, which can be backward or forward chaining. We conclude that the two ambassadors of rule-based languages are Prolog, a logic-based backward chainer, and OPS5, a forward chainer based on production rules.

Chapter 5: Hybrid Systems In this chapter, we consider hybrid systems that enable multiparadigm programming by integrating a rule-based language with the object-oriented one. This supports representation of rule-based knowledge in its own, more suitable paradigm instead of in the object-oriented language. We identify six issues with respect to integrating a rule-based and an object-oriented language. These issues are concerned with language support that is provided for activating rule-based programs from object-oriented

programs and the other way around. We categorise more than 50 hybrid systems and thoroughly investigate more than a dozen of those with respect to the six language integration issues. For each of these issues, the different approaches are generalised, discussed and illustrated. From the survey we conclude that there are two integration philosophies which coincide with the kind of rule-based language used (Prolog-like or OPS5-like). Furthermore, we observe that the hybrid systems' approaches to addressing language integration usually do not obtain obliviousness at the language level: features inherent to the one paradigm leak through to the other and vice versa.

Chapter 6: Model of Hybrid Composition and Hybrid Aspects This chapter introduces our approach which is a first foray of aspect-oriented programming in the context of hybrid systems. As such, we present a language-independent model which is a combination of hybrid composition and hybrid aspects. First, we indicate how both kinds of aspect-oriented programming approaches, one composition-based and one aspect-based, complement each other. The novelty of our approach is that it considers two base languages instead of one: a rule-based and an object-oriented language. In both cases, we consider basic features of these kinds of languages. Both hybrid composition and hybrid aspects consider the same dynamic join point model, which consists of traditional join points for object-oriented languages, extended with join points for different kinds of rule-based languages. Hybrid composition integrates rule-based and object-oriented programming employing a general correspondence and unambiguous match between join points and behaviour activation. In order to achieve this, a linguistic symbiosis between these elements is required. Hybrid aspects also integrate rule-based and object-oriented programs using join points and behaviour activation, but in an entirely different manner: a hybrid aspect language is provided for expressing hybrid pointcuts and hybrid advice. This language merges object-oriented and rule-based language features, which are described using an abstract grammar of our hybrid aspect language.

Chapter 7: Hybrid Composition and Hybrid Aspects in OReA In this chapter we present proof-of-concept implementations of hybrid composition and hybrid aspects for integrating several concrete rule-based languages and the object-oriented language Smalltalk. The implementations are developed in our atelier, *OReA*. This is initially outfitted with the existing hybrid system SOUL, which integrates Prolog and Smalltalk, and the aspect-oriented programming approach AspectS for Smalltalk. First, we implement two additional rule-based languages, reusing parts of the implementation of SOUL: a logic-based forward chainer and a production rule forward chainer. Hybrid composition is implemented using the reflective capabilities of Smalltalk, whereas hybrid aspects build upon AspectS. This chapter provides small yet concrete examples for illustrating the implementation of our approach.

Chapter 8: Evaluation The goal of the evaluation chapter is to investigate if our approach fulfils the criteria we set forth earlier. These criteria are distilled from issues that arise when integrating rule-based and object-oriented programs and languages. Although aspect-oriented programming is still relatively new, the general acceptance and awareness of its ability to ensure obliviousness of object-oriented programs is growing steadily. Therefore, in this evaluation, we concentrate on the other, and more original side of obliviousness at the program level: making rule-based programs oblivious to being integrated with object-oriented *and other* rule-based behaviour. In this regard, we select a case study that has a

considerable and sufficiently complex rule-based part: an application for semi-automated scheduling in domains such as television broadcasting and the educational system.

Chapter 9: Conclusions This chapter concludes this dissertation with a summary and a discussion of future work.

Chapter 2

Rule Objects

This chapter can be regarded as a first and gentle introduction to rule-based knowledge using a specific instance: business rules. Later on in this dissertation (chapter 8), we present more sophisticated instances of rule-based kowledge. Moreover, at this point, we only use concepts and techniques from state-of-the-art object-oriented software development for separating and integrating rules. We start with introducing the running example of this dissertation (section 2.1). In the next section we explain how rules can be represented with the *Rule Object Pattern* (section 2.2). Then, we introduce some object-oriented patterns that can be used for integrating rule objects with object-oriented functionality in a flexible way (section 2.3). However, the pattern-based solution has a number of shortcomings. Hence, an important part of this chapter is a section on the issues we identify, related to integrating object-oriented functionality and rule-based knowledge (section 2.4). This chapter ends with a summary (section 2.5).

2.1 Running Example: E-Commerce

Throughout this dissertation, a modest e-commerce application is used as a running example. It is a simplified online store for business-to-customer sales of books and cd's. We also introduce some simple business rules in this application.

2.1.1 Basic Functionality

The class diagram of the basic functionality of the application is shown in figure 2.1. Most of the examples in this dissertation are based on this diagram or show variations when needed.

The online store itself is modelled with the class `OnlineStore`, which refers to all the `Customer` objects and `Product` objects in the store. The `Accounting` class embodies the accounting department which holds previous orders and can determine credit rating of customers. When a customer shops, he or she puts products in his or her `ShoppingCart` object. A customer is checked out when the message `checkoutcustomer` is sent to the store. As a result, the contents of the shopping cart is assembled in an `Order`[1]. The total price of the order is calculated by the method `calculateTotalPrice()`. It adds all the prices of the individual products, obtained by sending `getPrice()` to the products.

Other functionality besides shopping and checking out is the ability to display products, which is used by the user interface when a customer is browsing products, and support for returning products.

[1] For the sake of simplicity we do not consider quantities of objects

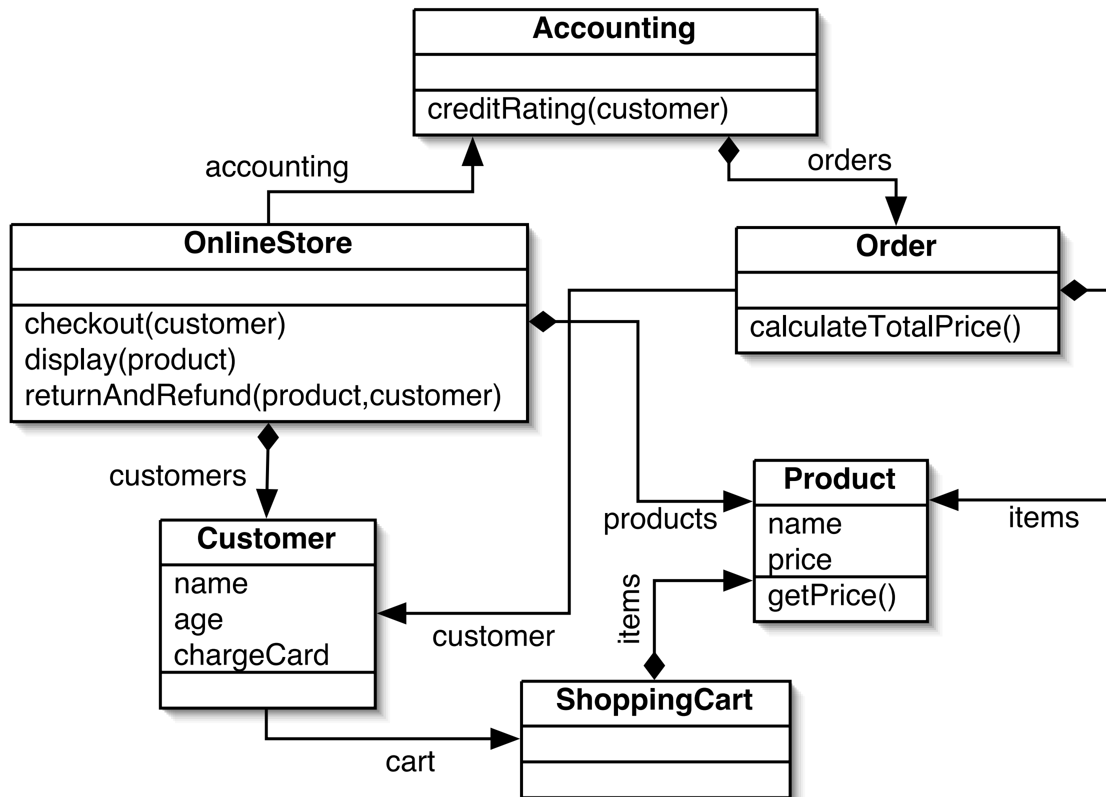


Figure 2.1: A class diagram of the basic functionality of an online store.

2.1.2 Rule-Based Knowledge

Rule-based knowledge is typically represented as a collection of rules. The *Business Rules Group* defines a rule as a statement that defines or constraints some aspect of the business, intended to to control the behaviour of the business [BRG, 2001]. Typically, a rule is applied at a certain event, which is a well-defined point in the execution of the object-oriented functionality.

Price Personalisation

Many of the existing approaches that separate explicit rules use the domain of e-commerce for illustrating and validating their ideas. We list some rules commonly found in e-commerce applications:

- policies for ordering [Grosof et al., 1999]:
  - refunds
  - returns
  - usage restrictions
  - lead time to place an order
  - cancelling orders
  - creditworthiness, trustworthiness, and authorisation

- policies for personalising links, structure, content and behaviour of web pages [Rossi et al., 2001b]
- policies that determine the control flow of online purchases [Arsanjani & Alpigini, 2001]
- policies found in online stores, e.g. *Amazon* (http://www.amazon.com) and *Proxis* (http://www.proxis.be):
  - recommendations
  - availabilities of products
  - returns
  - delivery rates
  - delivery restrictions

In this dissertation we use rule-based knowledge related to personalisation because this is an increasingly important and pertinent issue, as the special issue of the *Communications of the ACM* shows [CACMAdaptive, 2002]. Some examples of rules for price personalisation are:

- **XmasDiscountRule:** *a customer gets a 10% discount on Christmas day*
- **LoyalDiscountRule:** *a customer gets a 5% discount if he or she is loyal*
- **LoyalCustomerRule:** *a customer is loyal if he or she has a charge card and good credit rating*

The first two rules are price personalisation rules, but the third one is some kind of auxiliary rule. Indeed, *LoyalDiscountRule* depends on *LoyalCustomerRule*. Additionally, *LoyalCustomerRule* is also used when a product is returned to the store by a customer. The status of a customer influences the flexibility of the store in taking back or changing purchased products.

An example of an event where discount rules are applied is *when the price of a product is obtained*. Example events where *LoyalCustomerRule* is activated, are *when the price of a product is obtained* and *when a product is returned by a customer*. A more sophisticated event that could replace the first one, is *when the price of a product is obtained but only when checking out*. This last event rules out the event where the price of a product is obtained for displaying it while the customer is browsing.

Recommendations

Other important rule-based knowledge consists of recommending additional products to customers. We do not use this example in this and the following chapters. Once we introduce rule-based languages for expressing rules (chapter 4), it surfaces again.

Recommending products can have many forms and be applied at different points in the purchase process. For the sake of simplicity, we recommend products based on the products added to the shopping cart by the customer. Also, we do not recommend these additional products as much as simply adding them to the shopping cart without consulting the customer first.

- **BestsellerRecRule:** *add the current best-seller to the shopping cart if there is a book in the shopping cart*

- **BatteryRecRule:** *add batteries to the shopping cart if there is a product in the shopping cart that needs them*
- **WrapRecRule:** *add a gift wrapper to the shopping cart if there is a product in the shopping cart that needs it*

In our case, the event where the recommendation rules apply is *when the customer adds a product to his or her shopping cart.*

2.2 Rule Object Pattern

Once the rules are identified and represented explicitly on a conceptual level, this representation can be maintained at the implementation level as well. This means that each rule has to be represented explicitly and separately. As we will see later on, there exist dedicated *rule-based systems* which provide modular units for representing rules (chapter 4). For now, however, we make do with an object-oriented programming language for representing the rules. In other words, each rule is represented in the "modular unit" of the object-oriented programming language, i.e. an object.

The representation of rules presented in this section will still be relevant in the next chapter when we introduce aspect-oriented programming for integrating separated rule objects (chapter 3). However, once we start representing rule-based knowledge as real rules in a rule-based system, the rule object representation will be discarded (chapter 4).

2.2.1 Simple Rule Object

The *Rule Object Pattern* is an extension of *Adaptive Object Models* [Yoder & Johnson, 2002] and is devised for representing rules as objects [Arsanjani, 2001]. In its simplest form, a rule is implemented by a single class implementing the methods `condition()`, `action()` and `apply()`. Typically, the last method is invoked by a client which wishes to activate the rule. It tests the condition and performs the action if the condition evaluates to true.

The Rule Object Pattern can become quite elaborate, employing any number of *design patterns* [Gamma et al., 1995]. For example, a properties object can be passed to a rule to parametrise all possible values that are used or manipulated in the condition(s) and action(s) of the rule. Furthermore, rule objects can be composed in a *Composite Pattern* to indicate a set of related rules that should be activated together. In order to control the ordering, exclusion and combination of rules, a *Strategy Pattern* is used. This is all shown in figure 2.2. Extra features that are not shown are objectifying condition and action, introducing *Composite* patterns, including results of conditions and actions and coordinating the whole by a *Mediator Pattern*.

2.2.2 Implementing Rule Objects

Implementations of the example rules presented in section 2.1 as classes in Java are presented in code fragments 2.2, 2.3 and 2.4. The first two rule objects have a common abstract superclass, which is show in code fragment 2.1.

The abstract class `DiscountRule` represents the basic properties and behaviour of all price personalisation rules. It implements the method `applyprice`, which tests the condition and performs the action if the condition evaluates to true or returns the standard price of a product if it does not. Since each condition can be different for each price personalisation rule, the method `condition()` is abstract. `DiscountRule` also provides a default

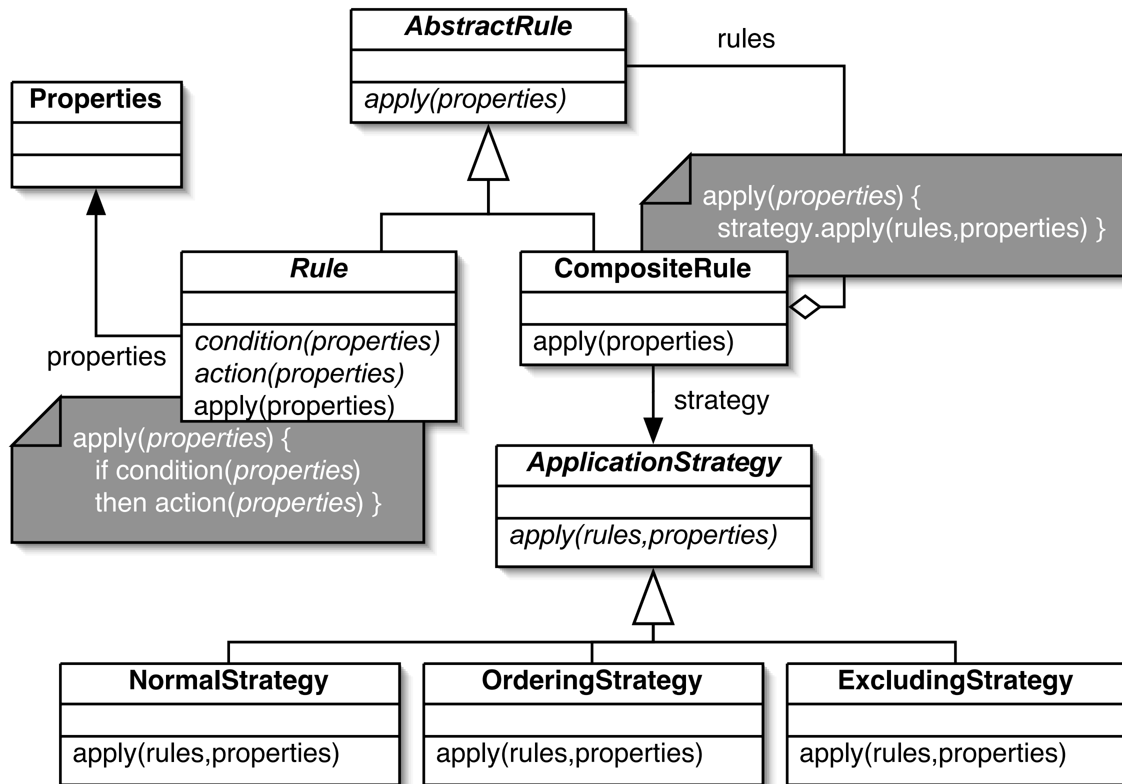


Figure 2.2: An abstract and concrete price personalisation Rule Object.

implementation for the method `action(price)` by subtracting the percentage attribute from the standard product price.

The concrete subclasses `XmasDiscountRule` and `LoyalDiscountRule` provide concrete implementations for the condition, checking if it is Christmas and checking if the customer is loyal, respectively. They also initialise the inherited `percentage` attribute with the correct discount percentage. The second rule object provides a new `apply` method since it requires an extra parameter, a customer, for establishing if a discount is warranted in the condition.

One could argue that a rule should be as declarative as possible and state what is true given certain conditions instead of what should be done under certain conditions. In that sense, a discount rule should not calculate the final price by subtracting a percentage, but rather infer that a certain discount percentage "is true" under certain conditions. The code that integrates the discount rules with the object-oriented functionality or an anticipated method in the functionality, is then responsible for doing the actual calculation. Here, we included the calculation in the rules for the sake of simplicity. In a later chapter, more declarative representations are introduced as well.

The implementation of *LoyalCustomerRule* also requires a customer object for checking the condition and performing the action.

2.3 Integration Patterns

Patterns for Personalisation enable the flexible integration of the rule-based knowledge represented as rule objects and the object-oriented functionality [Rossi et al., 2001a]. The

```
abstract public class DiscountRule{
  private Float percentage;
  abstract public boolean condition();
  public Float action(Float price){
    return(new Float(price-price*percentage/100));}
  public Float apply(Float price){
    if (condition()){
      return action(price);
    else{return price};}}}
```

Code Fragment 2.1: Implementation of an abstract discount rule object in Java.

```
public class XmasDiscountRule extends DiscountRule{
  XmasDiscountRule(){
    percentage = 10;}
  private boolean isChristmas(){...}
  public boolean condition(){
    return isChristmas(); }}}
```

Code Fragment 2.2: Implementation of the Christmas discount rule object in Java which inherits from the abstract discount rule object.

```
public class LoyalDiscountRule extends DiscountRule{
  LoyalDiscountRule(){
    percentage=5;}
  public boolean condition(Customer c){
    return c.isLoyal() }
  public Float apply(Float price, Customer c){
    if (condition(c)){
       return action(price);
    else{return price}; } }}
```

Code Fragment 2.3: Implementation of the loyal customer discount rule object in Java which inherits from the abstract discount rule object.

```
public class LoyalCustomerRule{
  public boolean condition(Customer c){
    return c.hasChargeCard() && c.hasGoodCreditRating(); } }
  public void action(Customer c){
    c.setLoyal(true);}
  public void apply(Customer c){
    if (condition(c)){
      action(c); } } }
```

Code Fragment 2.4: Implementation of the loyal customer rule object in Java.

basic idea is shown in figure 2.3, which integrates the basic functionality of figure 2.1 with only the *XmasDiscountRule*.

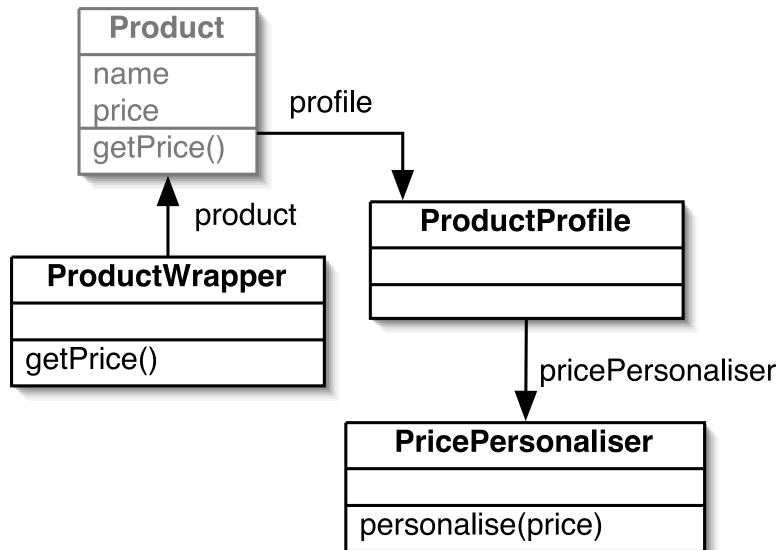


Figure 2.3: A class diagram of the extension of the basic functionality of the online store for integration with the Christmas discount rule object.

The extended design assumes that all price personalisation will occur at the event *when the price of a product is retrieved*, which is any invocation of `getPrice()`. `ProductWrapper` intercepts these calls and delegates them to a `PricePersonaliser` via a `ProductProfile`. This last class possibly holds different personalisers, such as a `PricePersonaliser` and a `ContentPersonaliser` for personalising the representation of product information for example (not shown in the figure). The use of explicit rules is facilitated because they are collected in the corresponding personalisers. Different profiles may require different sets of rules, which can be customised in the personalisers.

The methods `getPrice()` in `ProductWrapper` and `personalise(Float price)` in `PricePersonaliser` are implemented in Java:

```
getPrice(){
  return product.profile.pricePersonalizer.personalise(product.getPrice())}

personalise(Float price){
  XmasDiscountRule r = new XmasDiscountRule();
  r.apply(price);}
```

2.4 Program Integration Issues

In the previous section we presented some object-oriented programming techniques for integrating rule objects with object-oriented functionality. This approach attempts to minimalise the dependencies between the rule objects and the basic functionality: it shifts the dependency of the basic functionality on the rule objects to a series of intermediate objects (wrapper, personaliser, ...). However, when further integrating the basic object-oriented functionality with the other example rules which are activated at different events,

we quickly run into some problems. As a result of investigating these problems, we come up with a number of *program integration issues* that have to be dealt with when integrating rule-based knowledge and object-oriented functionality. Figure 2.4 illustrates the integration issues. We also discuss these issues in [D'Hondt et al., 1999] [Cibrán et al., 2003a] [Cibrán et al., 2003b]. In the remainder of this section, we explain the identified integration issues.

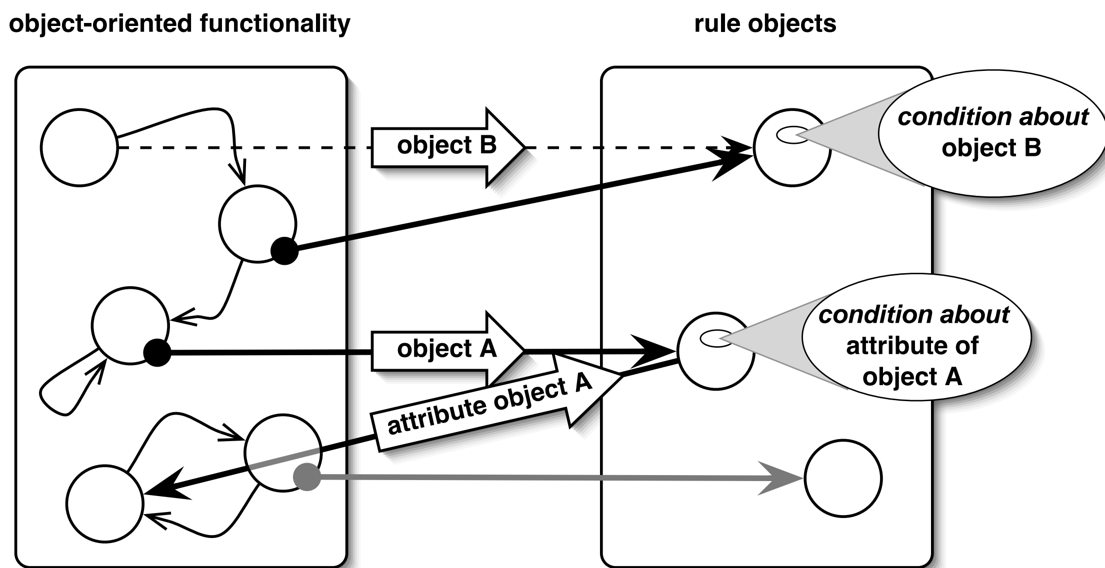


Figure 2.4: Dependencies between object-oriented functionality and rule-based knowledge.

2.4.1 Dependency

Since rules are activated at events in the object-oriented functionality, the latter depends directly on the former. However, unanticipated changes to the activation of rules can be required, such as activating a new rules at an event or activating rules at a new event in the object-oriented functionality. Because of the dependency of the object-oriented functionality on the rules, these unanticipated changes require invasive changes to be made manually in different places of the object-oriented functionality.

Even the design proposed in figure 2.3, which uses wrappers and personalisers to eliminate the direct dependency of basic functionality on rule objects, poses problems. For each new event, a wrapper has to be introduced for the object(s) that embody the event(s). This introduces a large number of auxiliary objects which become hard to manage. This is a known problem and discussed in [Nordberg III, 2001] and [Hannemann & Kiczales, 2002]. Moreover, we explain later on that adding rules to the same event by including them in an existing personaliser, is not so straightforward either.

Figure 2.4 illustrates the different kinds of dependencies between object-oriented functionality and rule-based knowledge. We explain these in terms of the level of coupling, a measure of interconnection among software parts [Stevens et al., 1974]. We identify three forms of coupling when integrating object-oriented functionality and rule-based knowledge.

Data Coupling

Rules typically express conditions about objects, which are passed to the rules when the object-oriented functionality activates rules at certain events. This is a low level of coupling, *data coupling*. This is shown in figure 2.4, where a rule object is activated (black arrow) and an object (*object A*) is passed for use in the rule object's condition.

However, even low coupling such as data coupling can complicate change. When the rule-based knowledge evolves or other rules have to replace it, other input values are possibly required from the object-oriented functionality. The problem is that the new required object might not be available in the scope of the event. This situation is illustrated in figure 2.4: a rule object is activated which requires a certain *object B* to match it to its condition. However, this particular object is not available in the scope of the event where the rule object is activated. It is available, though, at another event. The dotted arrow in figure 2.4 denotes that *object B* should be retrieved from the event where it is available and passed to the rule object. Hence, the change in rule-based knowledge propagates to the object-oriented functionality.

For example, take *LoyalDiscountRule* and *LoyalCustomerRule* which are both activated when `getPrice()` is called. Both rules need the customer who is buying the products as input. The method that calls `getPrice()` (in the control flow of `checkout(Customer c)`) is `calculateTotalPrice()` defined in `Order`. The customer is available in an order. A typical solution to this problem is passing the customer object along the control flow from where it is available to where it is needed. This requires adding an extra parameter to all the methods in the control flow. In the case of this example, a method `getPriceFor(Customer c)` replaces `getPrice()` and `personalise(Float p, Customer c)` replaces `personalise(Float p)`.

Again, this single issue introduces code that as scattered across the implementation which cannot be encapsulated using object-oriented programming techniques. This example also illustrates that adding new rules to the construction of personalisers is not always so trivial. A smooth change is possible when the new rules require the same or less input than is anticipated. The same goes for all other approaches that represent rule-based knowledge separately.

Control Coupling

We identify another form of tight coupling, *control coupling*, when the activation of rule-based knowledge depends on the run-time properties of the object-oriented functionality. More sophisticated events embody conditional activation of rules, and we refer to these as *dynamic events*. This is illustrated in figure 2.4 by the grey arrow.

A typical example is that rules are activated after the "try password" functionality has been activated three times. Another example is the event *when the price of a product is obtained but only when checking out* (section 2.1). This event denotes calls of the method `getPrice()` as before, but only if this method is called in the control flow of the `checkout(Customer c)` method. This rules out the situation where `getPrice()` is called when the customer is browsing products and their information is displayed by the `display(Product p)` method.

Typically the conditional part of the event is implemented by adding a flag which is set to true if a checkout process is started. An extra conditional evaluates this checkout flag before personalising the price of a product. This practice again results in code related to this issue being scattered, which means one has to be careful to consider all the possible places it occurs when adapting it.

Content Coupling

Many approaches for separating rule-based knowledge allow it to access attributes of the objects directly, which is the worst form of coupling: *content coupling*. This is illustrated in figure 2.4, where the rule accesses the object (black arrow) and the *attribute of object A* is returned.

2.4.2 Adaptation

When integrating two programs, incompatibilities between them due to evolution necessitate adaptations. Hence, integration consists of more than a simple call from rule-based knowledge to activate object-oriented functionality or the other way around. The programs are extended with some kind of ”glue code” for converting or manipulating the results of behaviour activation. Usually, approaches for separating rule-based knowledge require the adaptations to be expressed as extensions to objects. Hence, the object-oriented functionality is cluttered with code to accommodate each of its integrations with different rule-based knowledge. Figure 2.4 represents the adaptations with black dots. In the case of conditional integration depending on run-time properties, the adaptation is also conditional, hence the grey dot in figure 2.4.

2.4.3 Combining Rules

A final challenge when activating rules in object-oriented functionality, is that more often than not more than one rule is activated at the same event. This can be solved by using combination strategies as depicted in the diagram of the Rule Object Pattern in figure 2.2. Another approach of ordering rule activations is nesting wrappers: when calling a method on the outer wrapper it will activate the first rule and delegate to the object it refers to, which can be another wrapper that activates a second rule and so on.

Note that rule-based languages, presented in chapter 4, represent rules in dedicated language constructs and manage the flow of rules automatically. Therefore, the responsibility for dealing with this issue shifts from the integration to the representation of rule-based knowledge. We mention it here because the next chapter illustrates how existing aspect-oriented approaches deal with the integration of object-oriented functionality and rule objects. In this situation, combination of rules has to be considered explicitly. This exercise also illustrates the usefulness of rule-based languages for representing rules.

2.5 Summary

This chapter provides a gentle introduction to the use of explicit and separated rule-based knowledge. It illustrates the challenges and issues involved with as little technological overhead as possible, since only object-oriented programming techniques are used. This also serves to show the shortcomings of only using object-oriented programming techniques for integrating rule objects with object-oriented functionality.

More specifically, we presented a small example of an e-commerce example which consists of some basic object-oriented functionality and a few simple rules. This example will be used throughout this dissertation. The most important part of this chapter is a review of the integration issues, since we distil a set of criteria from them that approaches for dealing with integration of separated rule-based knowledge and object-oriented functionality must meet.

Chapter 3

Aspect-Oriented Programming

The purpose of this chapter is to introduce the ideas of aspect-oriented programming and show how some selected approaches provide concrete support for addressing the integration issues presented in the previous chapter (chapter 2).

The first section presents the rationale behind the principle of aspect-oriented programming (section 3.1). In the following section we discuss two important properties of aspect-oriented programming which indicate that it is indeed a suitable approach for dealing with the integration issues (section 3.2). Next, we argue our selection of aspect-oriented approaches (section 3.3).

In order to balance the rather abstract account in the first three sections of this chapter, we present three concrete aspect-oriented programming approaches in the next three sections, *HyperJ* (section 3.4), *AspectJ* (section 3.5) and *JAsCo* (section 3.6). We illustrate each approach using the example e-commerce application, which additionally serves to show the abilities of each approach with respect to the integration issues. Finally, we conclude this chapter with a discussion of the different approaches (section 3.7).

3.1 Separation of Crosscutting Concerns

A software application involves many and heterogeneous concerns. By concerns we refer to areas of interest in the system, which can materialise in all phases of the development cycle. Typically, concerns can range from high-level notions like security and quality of service to low-level notions such as caching and buffering. To deal with all these heterogeneous concerns in a software application *separation of concerns* is fundamental: it refers to the ability to identify, encapsulate and manipulate only those parts of the software that are relevant to a particular concept, goal or even development team [Parnas, 1972]. This allows a developer to decompose the software in parts that embody these naturally occurring concerns rather than letting the development tools impose a separation of the software in artificial parts.

When considering this phenomenon at the implementation level, we observe that programming languages are the development tools. Generally, programming languages provide a dominant decomposition mechanism, such as procedures or objects, for encapsulating concerns. However, this dominant decomposition mechanism is not always suitable for capturing and representing all kinds of concerns that can be found in a software application. Typically, there exist "system-wide" concerns that cut across the natural modules of the application and cannot be encapsulated by the dominant decomposition mechanism of a language [Kiczales et al., 1997]. This problem is referred to as *the tyranny of the dominant decomposition* [Tarr et al., 1999]. Well-known examples of *crosscutting concerns* in object-

oriented programs are systemic properties or behaviours such as logging, error recovery and synchronisation. These concerns cut across the object structure because they cannot be encapsulated in objects.

Aspect-oriented programming focuses on mechanisms for achieving a better separation of crosscutting concerns in software applications at the implementation level. Consider a program implemented in a standard procedural or object-oriented programming language. We refer to such a program as a *base program* and the language as a *base language*. A crosscutting concern typically affects this base program in many, non-local places. Aspect-oriented programming approaches provide mechanisms for expressing such crosscutting concerns in an encapsulated way and separately from the program they affect. Most approaches but not all provide a new kind of abstraction for representing crosscutting code in an encapsulated way.

Many adopt the terminology of *AspectJ* [Kiczales et al., 2001], probably because it is one of the first aspect-oriented approaches and one of the best known. Additionally, many newer approaches are based on the AspectJ approach, for the same reasons. However, there are approaches that provide very different support for encapsulating crosscutting concerns.

According to AspectJ terminology, the places that are affected by a crosscutting concern are referred to as *join points*. These can be dynamic — points in the execution of a program — or static — elements of a program definition. AspectJ-like approaches provide an new kind of abstraction, an *aspect*, for encapsulating a crosscutting concern. Typically, the aspect represented in a general-purpose aspect language. Integrating the base program with an aspect program at the specified join points is a process called *weaving*.

We use this terminology throughout the dissertation, unless it is not compatible with a particular aspect-oriented approach being discussed.

3.2 Obliviousness and Quantification

Two distinguishing characteristics of aspect-oriented programming systems are *quantification* and *obliviousness* [Filman, 2001]:

- Quantification relates to support for specifying many, non-local and heterogeneous join points. Many aspect-oriented systems support *dynamic* quantification, which weaves input programs depending on run-time properties [Filman & Friedman, 2000].
- Obliviousness, on the other hand, is the idea that a program does not have to anticipate its being integrated with other programs. It is the opposite of the "don't call us, we'll call you"-principle: it is the "we won't call you, you'll have to call us"-principle. This is because the programs do not take care of the integration by calling each other directly. Rather, there is an *integrator* which makes the integration explicit. As a result, the program being integrated do not depend on each other directly, but the explicit *integration* is dependent on the programs instead [Nordberg III, 2001].

Although the primary and best-known goal of aspect-oriented programming is modularising crosscutting concerns — hence the need for quantification — it is also a very powerful means for achieving minimal dependencies between modules or classes. This is mainly due to the other characteristic of aspect-oriented approaches: obliviousness. A program that is oblivious to being integrated with another program can be reused in other integrations, without it having to carry around the extra baggage of all the integrations it is part of.

Both [Hannemann & Kiczales, 2002] and [Nordberg III, 2001] discuss the advantages of dependency inversion with aspect-oriented programming in the context of design patterns.

3.3 Selecting Aspect-Oriented Approaches

Although there are many different aspect-oriented approaches, each with their own particular technique for separating crosscutting concerns, they all ensure obliviousness of the base programs. Most of the existing approaches use an object-oriented programming language as base language. This means that, currently, object-oriented programs can be integrated with each other while remaining oblivious to the integration.

In this dissertation, we do not wish to contribute to techniques for achieving obliviousness. Instead, we want to transfer existing aspect-oriented techniques for achieving obliviousness to hybrid systems. The goal is to achieve obliviousness at the program and language level of both object-oriented and rule-based programs and languages. Since in this chapter and the previous, we consider a purely object-oriented development environment and not a hybrid one, we run ahead of our story. Nevertheless, since we are going to experiment with existing aspect-oriented approaches in this chapter, we need to indicate the reasons for selecting the ones we did.

First of all, integrating two independent, meaningful programs — one implementing core functionality and the other rules — is cumbersome if done manually, i.e. if an aspect has to be defined for each integration point. One kind of aspect-oriented approach is based on *composition*. It composes two base programs according to a global composition strategy. The idea is that elements of both base programs are identified that map onto each other. This results is a default, global and fully automated integration of two base programs. The best-known composition approach is *Multi-Dimensional Separation of Concerns* [Tarr et al., 1999] and the concrete tool *HyperJ* [Ossher & Tarr, 2001] for Java. Part of our aspect-oriented approach for integrating rule-based knowledge and object-oriented functionality is based on composition. Therefore, in section 3.4, we present HyperJ and show how it separates object-oriented functionality and rule objects.

Secondly, one of the integration issues presented in the previous chapter is the need to adapt either the core functionality or the rules when they are integrated. As such, an explicit construct, such as an aspect, for encapsulating the integration code — including the adaptation — is desired. Since the integration code expresses activation of the base programs and manipulates their values, a general-purpose aspect language is required. Moreover, we argue later on in this dissertation, that such an aspect language is also the key to achieving obliviousness at the language level.

There are many aspect-oriented approaches that are based on the pointcut/advice model and provide a general-purpose aspect language. Some of these approaches extend the pointcut/advice model in order to apply aspects in other development environments. Examples are *JAsCo* [Suvée et al., 2003] and *JAC* [Pawlak et al., 2001], which both provide aspects for component-based development. Other approaches improve the original pointcut/advice approach in order to obtain reusable aspects and composition of aspects. In AspectJ, the application context of an aspect is hard-coded in the aspect itself, which severely limits reuse of the aspect in other contexts. *Aspectual Components* is one of the first approaches to remedy this, by introducing explicit connectors that connect aspects to a specific context [Lieberherr et al., 1999]. Newer approaches based on Aspectual Components are *Aspectual Collaborations* [Lieberherr et al., 2003] and *Caesar* [Mezini & Ostermann, 2003]. Since, to our knowledge, we are the first to develop a pointcut/advice model for hybrid systems

and a *hybrid* aspect language, we decide not to consider advanced issues such as aspect reusability and aspect composition. Instead, we opportunistically select the "mother" of all pointcut/advice approaches, AspectJ. It is used as a basis for the second part of our aspect-oriented approach for integrating rule-based knowledge and object-oriented functionality. In section 3.5, we present HyperJ and show how it separates object-oriented functionality and rule objects.

There are two other important aspect-oriented approaches that we wish to mention: *Adaptive Programming* and *Composition Filters*. They are two of the first aspect-oriented approaches. As such, they played an important role in shaping the state of the art in aspect-oriented programming, and still do.

Adaptive Programming provides *adaptive methods* for supporting the *Law of Demeter* in object-oriented programs [Lieberherr et al., 2001]. Adaptive methods consist of a traversal strategy and an adaptive visitor. The former describes how to reach the objects that participate in a certain operation in a high-level way. The latter encapsulates the code related to the operation and visits the participating objects according to the traversal strategy. As such, the adaptive method can be reused for other object structures. Although hiding object structure can be an issue in integrating rule-based knowledge and object-oriented functionality, this specific concern is not the first priority at this point.

The Composition Filters approach provides *filters* for extending existing classes non-invasively, i.e. without having to change the original implementation of the classes. Filters intercept incoming and outgoing messages of objects and "attach" concerns to the objects such as inheritance and delegation, synchronisation, real-time constraints and interobject protocols [Bergmans & Aksit, 2001]. In order to express these concerns, a number of predefined filter types are available. As such, the Composition Filters approach supports *intra-object crosscutting*: different messages of a single object can be adapted with a single filter that modularises the crosscutting code, instead of implementing the code in each of the corresponding methods. More recently, support for defining *inter-object crosscutting* has be added to the Composition Filters approach. The idea is to use *filter modules*, which contain groups of filters and related definitions, and compose them with concerns using *superimposition*. The superimposition specifications describe the locations within the program where concern behavior is to be added. The Composition Filters approach unifies classes, filter modules and superimposition into concerns. As such, it adopts a single abstraction as the major module concept; the concern abstraction. As a result of the unification into concerns, the model does not distinguish between aspect and base module abstractions, which introduces symmetry. The implementation parts of the concern can be expressed in a variety of languages. The Composition Filters approach can be used for integrating rule-based knowledge and object-oriented functionality, implemented in the corresponding programming language of a hybrid system. Indeed, implementation parts of a concern can be implemented in object-oriented or rule-based languages. Even possibly hybrid integration code can be expressed as an implementation part of a concern, implemented in a hybrid language that combines features of both aforementioned languages. However, the concept of filters is conceived for the object-oriented programming paradigm and considers object interfaces, and incoming and outgoing messages. This concept will have to be reviewed entirely when considering filters for rule-based languages.

3.4 HyperJ for Integrating Rule Objects

HyperJ supports the principle of Multi-Dimensional Separation of Concerns for software applications implemented in Java. The idea behind this approach is that simultaneous encapsulation of all kinds (dimensions) of concerns in a software application — possibly overlapping or interacting — should be supported as well as on-demand remodularisation to encapsulate new concerns at any time. Multi-Dimensional Separation of Concerns argues that providing alternative modules for encapsulating crosscutting concerns — as other aspect-oriented programming approaches usually do — will only be suitable for some concerns and in some contexts. Conversely, this approach starts from existing modules or units for structuring the concern space, such as modelling elements and diagrams, classes and interfaces, methods, requirement specifications and so on. Multi-Dimensional Separation of Concerns provides a means for structuring these units to reflect the natural concerns in the concern space. The proposed structuring mechanisms are the following:

- a *hyperslice* encapsulates a concern
- a *hypermodule* specifies the integration of hyperslices

Hyperslices can be defined non-invasively when the encapsulation of some concern is not anticipated. Hyperslices are symmetric without one dominating the other. A hypermodule consists of a set of composition rules that specify how the hyperslices should be composed.

HyperJ supports this model at the implementation level, extending Java with mechanisms for structuring units of software instead of introducing a new language or formalism.

3.4.1 Dependency

Aspect-oriented programming approaches are originally conceived with implementation-level aspects in mind, such as synchronisation, error handling and logging. However, [Ossher & Tarr, 2001] shows that Multi-Dimensional Separation of Concerns can also applied to other kinds of concerns, such as business rules. The examples shown here are adapted from [Ossher & Tarr, 2001].

Figure 3.1 shows three different dimensions graphically, class, feature and rule, denoted by the axes. The *class dimension* can be viewed as representing the data concern. The *feature dimension* tends to reflect end-user concerns and perspectives. The *rule dimension* encapsulates rule-based knowledge, implemented as rule objects in this case. Note that in the *Product* class in the *Display* feature concern, the method *display()* uses the methods *name()* and *getPrice()*. However, since *Product* in this feature concern does not implement *name()* and *getPrice()*, they are added as abstract methods in order to make the feature concern *declaratively complete*. In figure 3.1, both *name()* and *getPrice()* are slanted to denote this.

In HyperJ, dimensions can be expressed on an existing class structure in a *concern mapping*. The concern mappings along the feature dimension in figure 3.1 are given in code fragment 3.1. Suppose that all classes of the e-commerce application are defined in the `ECOM` package. The first statement in this concern mapping expresses that all the classes in package `ECOM`, including `Product`, belong to the `Checkout` concern in the `Feature` dimension. Note that concerns and dimensions are not predefined, but can be chosen by HyperJ developers. The second statement expresses that any method with selector `display` of any class belongs to the `Display` concern in the `Feature` dimension. As such, the second statement overrides the first: the operation is only added to `Feature.Display`, not to `Feature.Checkout`. This concern mapping results in the creation of two hyperslices,

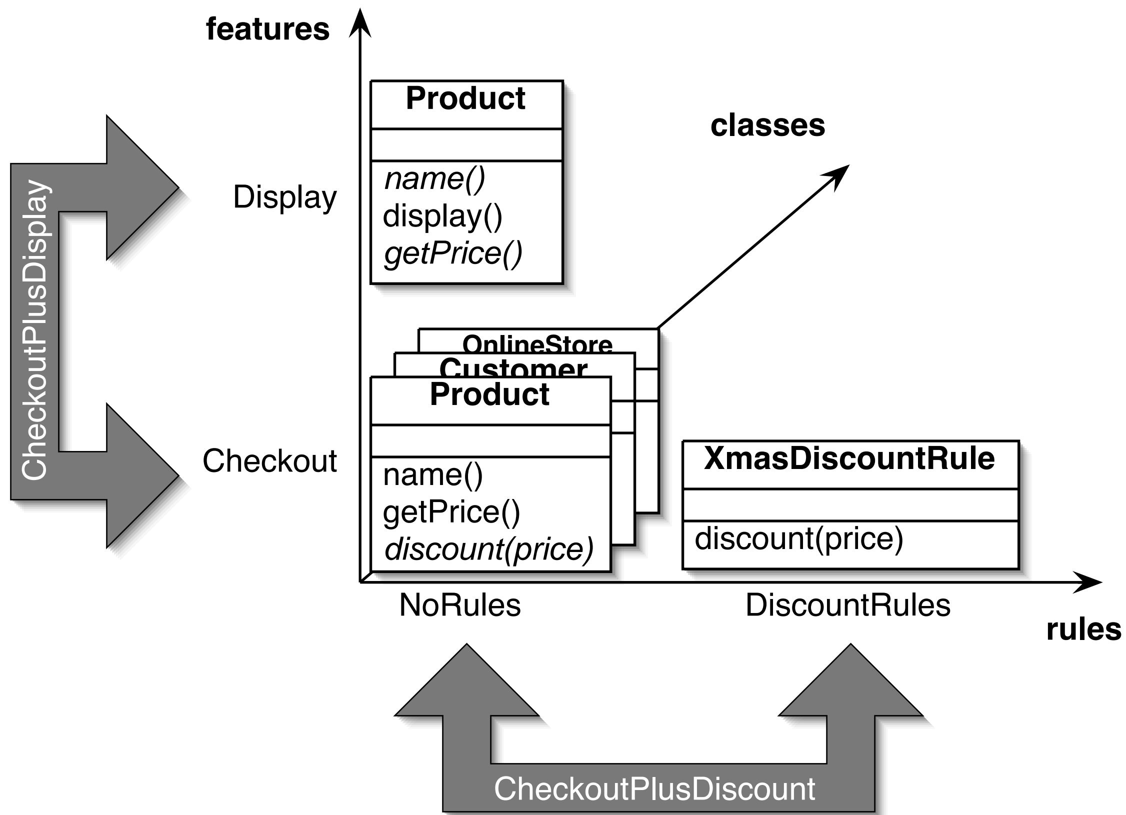


Figure 3.1: Classes of the e-commerce application along feature, rule and class dimensions (adapted from [Ossher & Tarr, 2001]).

`Checkout` and `Display`, each encapsulating its own class hierarchy. The classes in the first hierarchy contain all the methods in the original system, save for the method `display()` which is included in the second hierarchy.

```
package ECOM: Feature.Checkout
operation display: Feature.Display
```

Code Fragment 3.1: Concern mappings along the feature dimension in HyperJ (adapted from [Ossher & Tarr, 2001]).

Figure 3.1 also shows two integrations. First, consider the integration on the left-hand side. In order to integrate the two feature concerns, a hypermodule needs to be specified. This is shown in code fragment 3.2. This declaration first lists the hyperslices that are to be integrated, `Feature.Checkout` and `Feature.Display`, respectively. The integration rule is `mergeByName`, which states that units from both hyperslices with the same names are to be merged.

The other hypermodule shown in figure 3.1 is defined in code fragment 3.3. It makes sure that when `getPrice()` invokes `discount()` on `Product`, the corresponding method in `XmasDiscountRule` is invoked instead. Note that this method is the equivalent of the `apply(Float p)` method in the rule objects presented in the previous chapter.

```
hypermodule CheckoutPlusDisplay
  hyperslices:
    Feature.Checkout, Feature.Display;
  relationships:
    mergeByName;
end hypermodule
```

Code Fragment 3.2: Hypermodule `CheckoutPlusDisplay` in HyperJ (adapted from [Ossher & Tarr, 2001]).

```
hypermodule CheckoutPlusDiscount
  hyperslices:
    Rules.NoRules, Rules.DiscountRules;
  relationships:
    mergeByName;
    equate class Rules.NoRules.Product,
    Rules.DiscountRules.XmasDiscountRule;
end hypermodule
```

Code Fragment 3.3: Hypermodule `CheckoutPlusDiscount` in HyperJ (adapted from [Ossher & Tarr, 2001]).

It is clear that the two pairs of hyperslices that are integrated with the above hypermodules are oblivious to one another. As a result, no integration-dependent code needs to be added in order to prepare the hyperslices for a particular integration. They can be adapted and developed independently, and any incompatibilities can be taken care of in the hypermodule. Hypermodules have an entire arsenal of integration rules at their disposal such as bracketing, which controls the order of combining method bodies, overriding and renaming.

Note that there is an even simpler implementation, one without the extra rule object: the hyperslice `Rules.DiscountRules` contains the `Product` class and the method `discount(Float p)`. Again, this method stands for the application of the rule, which means checking the condition and performing the action. This requires a hypermodule which contains only the `mergeByName` statement.

3.5 AspectJ for Integrating Rule Objects

AspectJ is a general-purpose extension to Java that supports the pointcut/advice approach described earlier. AspectJ has been successfully used to encapsulate crosscutting concerns such as tracing, contract enforcement, display updating, synchronisation, consistency checking, protocol management and so on [Kiczales et al., 2001].

Aspect-Oriented Programming structures a software application in a base program, which addresses the dominant concern, and several separate *aspect programs*, which each address a different crosscutting concern. These aspect programs consist of a new module for encapsulating crosscutting concerns, an *aspect*. An aspect specifies a set of join points in the base program, which are described by a *pointcut*. Another part of the aspect is the *advice*, which consists of statements that should affect the base program at the join points described by the aspect's pointcut. The weaver combines the input programs, i.e. base and

aspect programs, into a single execution.

We conducted an elaborate experiment with AspectJ for integrating rule objects with basic object-oriented functionality, which is described in [Cibrán et al., 2003a].

3.5.1 Dependency

AspectJ supports dependency inversion which addresses data coupling. Events where rules are activated correspond to join points in AspectJ. Advice is used to specify code that should be executed at each of the join points of a pointcut and as a result can be used to activate the rules. AspectJ is able to express *dynamic* join points, which are a means for addressing control coupling. Moreover, AspectJ supports pointcut parameters, which allows us to deal with content coupling as well.

As before, the rules themselves will be implemented as rule objects. Aspects will be used for encapsulating the code that integrates object-oriented functionality with rules. However, the integration itself can be divided in reusable parts conceptually, so several aspects are used for implementing it. Typically, there we distinguish:

- an aspect expressing the activation of rule objects (data and control coupling), its name starting with `Event`
- an optional aspect for capturing unavailable objects to the event (content coupling), its name starting with `Capture`
- a last aspect that puts the previous ones together, its name starting with `Apply`

Data Coupling

As a first demonstration of how AspectJ is used, a new implementation of the application of `LoyalDiscountRule` is presented in code fragment 3.4. As opposed to the implementation with object-oriented patterns, the AspectJ implementation makes the object-oriented functionality oblivious of the rule being activated. The aspect defines pointcut `priceCalculation(Product p)` which captures each call to `getPrice()` and exposes the target of this method, an instance of `Product`. Note that this aspect does not define advice.

```
aspect EventPricePersonalisation{
  pointcut priceCalculation(Product p):
    call(Float Product.getPrice()) && target(p);}
```

Code Fragment 3.4: The aspect `EventPricePersonalisation` which describes the event at which discount rules are activated.

The only other aspect needed holds the rule to be applied and defines an *around advice* on the join points designated by the pointcut `priceCalculation(Product p)` (code fragment 3.4). This is shown in code fragment 3.5. An around advice replaces the original execution with the advice, with the option to proceed with the original execution. The pointcut exposes the instance of `Product` to the body of the advice where it is used to obtain the standard price (by calling `proceed(p)`) after which the resulting price is passed to the rule when it is applied (by calling `apply(price)`).

```
aspect ApplyXmasDiscount{
  static XmasDiscountRule rule;
  static public void setRule(XmasDiscountRule r){
    rule = r; }
  Float around(Product p):
      EventPricePersonalisation.priceCalculation(p){
    Float price = proceed(p);
    return rule.apply(price); }}
```

Code Fragment 3.5: The aspect `ApplyXmasDiscount` which activates `XmasDiscountRule`.

Control Coupling

We want to express the more sophisticated event *when the price of a product is obtained but only when checking out* in AspectJ. This event rules out the event where the price of a product is obtained for displaying it while the customer is browsing. Hence, this *dynamic* event depends on the control flow. The alternative implementation of the `EventPricePersonalisation` aspect is shown in code fragment 3.6. In addition to the first pointcut `priceCalculation(Product p)` a second pointcut needs to be specified, `priceCalcInCheckout(Product p)`. This one matches join points designated by the first pointcut and that are in the dynamic scope of join points matching the pointcut designated by the execution of `checkout(Customer c)`. This is done using the primitive pointcut designator `cflow`.

```
aspect EventPricePersonalisation{
  pointcut priceCalcInCheckout(Product p):
    call(Float Product.getPrice()) && target(p) &&
    cflow(execution(Float OnlineStore.checkout(Customer))); }
```

Code Fragment 3.6: An alternative implementation of the `EventPricePersonalisation` aspect which describes the *dynamic* event at which discount rules are activated.

Content Coupling

Both *LoyalDiscountRule* and *LoyalCustomerRule* require the customer who is currently purchasing the products as input. Since the customer is not available when the price of a product is obtained, we define another pointcut that denotes a join point which has this customer in its scope.

But first, consider the application aspect for activating the *LoyalDiscountRule* with the customer in code fragment 3.7. This aspect holds the rule and activates it at the join points denoted by the pointcut in the aspect `EventPricePersonalisation` (code fragment 3.4). What is different, is the attribute for holding the captured customer and a corresponding mutator. As such, `apply` can be invoked on the rule with the extra parameter as required.

The customer is set by the aspect `CaptureCustomer` in code fragment 3.8 after capturing it at a call of `checkout(Customer c)` to instances of `OnlineStore`. Because the aspect `ApplyLoyalDiscount` has state, it should be instantiated each time its customer is set, in other words, each time the aspect `CaptureCustomer` is activated at its join points. This is ensured by the `percflow` statement. Note that the statement `ApplyLoyalDiscount.aspectOf()` picks the correct aspect instance in which the customer should be set.

```
aspect ApplyLoyalDiscount percflow(CaptureCustomer.checkout(Customer)){
  static LoyalDiscountRule rule;
  static public void setRule(LoyalDiscountRule r){
    rule = r; }
  Customer customer;
  public void setCustomer(Customer c){
    customer = c; }
  Float around(Product p):
      EventPricePersonalisation.priceCalcInCheckout(p){
    Float price = proceed(p);
    return rule.apply(price,customer); } }
```

Code Fragment 3.7: The aspect `ApplyLoyalDiscount` which activates `LoyalDiscountRule`.

```
aspect CaptureCustomer{
  pointcut checkout(Customer c):
    call(public Float OnlineStore.checkout(Customer)) && args(c); }
  before(Customer c):checkout(c){
    ApplyLoyalDiscount.aspectOf().setCustomer(c);}}
```

Code Fragment 3.8: The aspect `CaptureCustomer` which captures the customer when he or she is checking out and exposes it.

3.5.2 Adaptation

Typically, a discount is a declarative expression which “infers” the discount percentage a certain customer is entitled to. Therefore, instead of actually calculating the discounted price, a discount rule should just return the discount percentage. This is shown later on in this dissertation, when rule-based languages are used for representing rules. Code fragment 3.9 shows an alternative implementation of the aspect which applies the `XmasDiscountRule`. The rule is activated with the message send `rule.discount()`, which returns the discount percentage. The actual calculation of the new price, taking the discount percentage into account, is also implemented in the advice in code fragment 3.9. It is apparent that this aspect adapts the rule to the object-oriented functionality.

```
aspect ApplyXmasDiscount{
  static XmasDiscountRule rule;
  static public void setRule(XmasDiscountRule r){
    rule = r; }
  Float around(Product p):
      EventPricePersonalisation.priceCalculation(p){
    Float price = proceed(p);
    return(new Float(price-price*rule.discount()/100));}
```

Code Fragment 3.9: An alternative implementation of the aspect `ApplyXmasDiscount` which activates `XmasDiscountRule` and *adapts* its result.

3.6 JAsCo for Integrating Rule Objects

JAsCo addresses aspect-oriented programming in the context of Component-Based Software Development [Suvée et al., 2003]. It is based on two existing aspect-oriented programming approaches: AspectJ and Aspectual Components. AspectJs main advantage is the expressiveness of its language to describe join points. However, AspectJ aspects are not reusable, since the context on which an aspect needs to be deployed is specified directly in pointcut part of the aspect definition. To overcome this problem, the concept of Aspectual Components was introduced. The rationale is that aspect-oriented programming means being able to express each aspect in terms of its own modular structure and as independent as possible of the base program. Hence, an aspect is defined with a set of abstract join points which are concretised when an aspect is combined with the base program. As such, JAsCo combines the expressive power of AspectJ with the aspect independency idea of Aspectual Components. Specific capabilities of JAsCo are:

- Aspects are described independent of a concrete context, making them highly reusable.
- JAsCo allows easy application and removal of aspects at run time.
- JAsCo has extensive support for specifying aspect combinations.

JAsCo is developed for the *Java Bean* component model. The JAsCo language itself stays as close as possible to the regular Java syntax and introduces two new concepts: *aspect beans* and *connectors*. An aspect bean is an extension of the Java Bean component that is able to specify crosscutting behaviour. Besides standard Java declarations, it contains one or more *hooks*, generic specifications of the application context of an aspect. A connector, on the other hand, instantiates one or more logically related hooks with a specific context. It is responsible for applying the crosscutting behaviour of the aspect beans and for declaring how several of these aspects collaborate.

A new, backward compatible component model is introduced that enables run-time loading and unloading of connectors.

We conducted an elaborate experiment with JAsCo for integrating rule objects with object-oriented functionality, which is described in [Cibrán et al., 2003b].

3.6.1 Dependency

Aspect beans can be used to implement rules and to specify the events at which they are activated in an abstract way. Hence, although an aspect bean combines rules and events in the same module, it is still able to maintain a separation between the two. Since different connectors can be defined to concretise the same hook, a rule can still be deployed in a variety of contexts.

An implementation of the *XmasDiscountRule* as an aspect bean in JAsCo is given in code fragment 3.10. The `XmasDiscountRule` aspect bean represents the rule (as in a rule object) and a hook. A hook specifies where the normal execution of the base program should be interrupted and what extra behaviour should be executed at that precise moment in time. In order to define when the functionality of a hook should be executed, the hook is equipped with at least one constructor that takes one or more abstract method parameters as input. These abstract method parameters are used for describing the abstract context of a hook. This generic specification of the context of an aspect makes rules reusable and as a result deployable in different contexts. The `XmasDiscountHook` specifies that its behaviour is deployable on every method with zero parameters that returns a `Float`. The constructor body

describes how the join points of a hook initialisation should be computed. In this particular case, the constructor body specifies that the functionality of the `XmasDiscountHook` should be performed whenever method is executed. The behaviour methods of a hook on the other hand, are used for specifying the various actions a hook needs to perform whenever one of its calculated join points is encountered. As in AspectJ, three behaviour methods are available: before, after and replace. The replace behaviour method of the `XmasDiscountHook` specifies that some discount is given, whenever the `isApplicable` method returns true. This method specifies a dynamic condition that is executed at a join point to check whether the connector is applicable.

```
class XmasDiscountRule {
  private Float percentage;
  public void setPercentage(Float p) {
    percentage = p;}
  private boolean isChristmas(){...}
  public boolean condition(){
    return isChristmas(); }
  public Float action(Float price) {
    return(new Float(price-price*percentage/100));}


  hook XmasDiscountHook {
    XmasDiscountHook(Float method()) {
      execute(method);}
    isApplicable() {
      return condition();}
    replace() {
      Float price = method();
      return action(price);}}}
```

Code Fragment 3.10: The aspect bean `XmasDiscountRule` which defines the rule and the hook `XmasDiscountHook`.

Connectors are used for instantiating one or more logically related hooks with a specific context (method or event signatures) and for specifying advanced aspect combinations. Connectors make it possible to deploy rules in a specific context. The connector `XmasDiscountDeployment` for the aspect bean `XmasDiscountRule` (code fragment 3.10) is defined in code fragment 3.11. This connector initialises the `XmasDiscountHook` with the `getPrice()` method defined in the `OnlineStore` class. After initialising this hook, the `XmasDiscountDeployment` sets the discount percentage and specifies the execution of the replace behaviour method.

3.6.2 Combining Rules

An advantage of having explicit connectors is the ability to combine, order or exclude rules that are activated together. JAsCo allows arranging the execution of a set of rules, by explicitly specifying the desired sequence in the connector. Whenever two or more hooks interfere, the order in which their behaviour must be executed is derived from the connector.

The connector in code fragment 3.12 activates the *XmasDiscountRule* aspect bean (code fragment 3.10) prior to the *LoyalDiscountRule* aspect bean (not shown). The initialisation

```
connector XmasDiscountDeployment {
  XmasDiscountRule.XmasDiscountHook h =
    new XmasDiscountRule.XmasDiscountHook(Float Product.getPrice());
  h.setPercentage(new Float(10));
  h.replace();}
```

Code Fragment 3.11: The connector `XmasDiscountDeployment` which instantiates the hook of the `XmasDiscountRule` aspect bean.

of the two hooks is similar to the initialisation in the `XmasDiscountDeployment` connector shown in code fragment 3.11. The last two statements activate the *XmasDiscountRule* and then the *LoyalDiscountRule*.

```
connector XmasAndLoyalDiscountDeployment {
  XmasDiscountHook xmasHook = new ...;
  LoyalDiscountHook loyalHook = new ...;
  xmasHook.replace();
  loyalHook.replace();}
```

Code Fragment 3.12: The connector `XmasAndLoyalDiscountDeployment` which instantiates the hooks of the `XmasDiscountRule` and `LoyalDiscountRules` aspect beans.

However, being able to specify the sequence in which the various rules are executed does not address all problems with rule combinations. Sometimes only one rule should be activated although more than one is applicable. An alternative combination is that one rule should be activated only when another is applicable. JAsCo addresses these advanced combinations with *combination strategies*. A combination strategy acts like a kind of filter that validates the list of applicable hooks, which are obtained at run time. Each specific combination strategy implements the `CombinationStrategy` interface, shown in code fragment 3.13. The interface itself only declares the `validateCombinations` method, which is used to describe the specific logic of a combination strategy. This mechanism of combination strategies allows maximum flexibility, as user-defined relationships between the various aspects can be specified.

```
public interface CombinationStrategy {
  public HookList validateCombinations(HookList aHookList);}
```

Code Fragment 3.13: The `CombinationStrategy` interface.

The exclude combination strategy in code fragment 3.14 specifies a combination strategy where the behaviour of `hook B` cannot be executed whenever `hook A` is encountered.

Code fragment 3.15 shows an alternative connector that deploys both *XmasDiscountRule* and *LoyalDiscountRule*. It specifies an exclude combination strategy between both rules. As a result, whenever the *XmasDiscountRule* is applied, the behaviour of the *LoyalDiscountRule* is ignored.

Most aspect-oriented technologies, AspectJ included, do not allow sophisticated control for instantiating, initialising and executing aspects, as this is done implicitly when the aspect is woven into the core of the base application. The JAsCo system improves these techniques, as the instantiation of an aspect with a specific context is described explicitly

```
class ExcludeCombinationStrategy implements CombinationStrategy {
  private Object A;
  private Object B;
  public ExcludeCombinationStrategy(Object hookA, Object hookB) {
    A = hookA;
    B = hookB;}
  public HookList validateCombinations(HookList aHookList) {
    if (aHookList.contains(A)) {
      aHookList.remove(B);}
    return aHookList;}}
```

Code Fragment 3.14: A combination strategy which excludes the execution of `hook B` whenever `hook A` is encountered.

```
connector XmasAndLoyalDiscountDeployment {
  XmasDiscountHook xmasHook = new ...;
  LoyalDiscountHook loyalHook = new ...;
  ExcludeCombinationStrategy strategy =
      new ExcludeCombinationStrategy(xmasHook,loyalHook);
  addCombinationStrategy(strategy);
  xmasHook.replace();
  loyalHook.replace();}}
```

Code Fragment 3.15: An alternative connector `XmasAndLoyalDiscountDeployment` which uses an exclude combination strategy.

in the connector. As a result, every instantiated aspect can be accessed as being a first class entity. Also, the execution of the behaviour of the rules is specified explicitly in the connector, which allows even more fine-grained control.

3.6.3 Dynamic Reconfiguration of Rules

Rules tend to evolve frequently and evolution without interrupting the software's execution should be supported. Consequently, we want to add, edit and remove rules at run-time. Most current aspect-oriented programming technologies, however, do not support run-time reconfiguration of aspects, as the deployment of an aspect within the system is rather static. This is mainly because an aspect loses its identity when it is woven into the base program. JAsCo solves this issue by also providing a run-time separation between the aspects and the base program. This way, JAsCo aspects remain first-class entities when they are deployed and their logic is not weld together with the basic functionality of the application. This property of the JAsCo system is a valuable concept in the context of integrating rules with object-oriented functionality, because this run-time separation together with the new component model allows dynamic reconfiguration of rules, without the need to shut down business-critical applications.

3.7 Discussion

Let us briefly summarise first. This chapter introduces aspect-oriented programming and three concrete approaches: HyperJ, AspectJ and JAsCo. These approaches are explained and illustrated using the e-commerce example application. Moreover, we attempted to solve the integration issues presented in the previous chapter (section 2.4) with each of the approaches. Of course, not all approaches deal well with all of these issues, hence, for each approach we only discussed the particular issues for which it has an expressive and elegant solution.

This section discusses the different approaches and compares how they deal with the integration issues.

HyperJ is distinctly dissimilar compared to the other two approaches. The reason is that HyperJ is foremost a decomposition and composition tool, and secondly an aspect-oriented programming approach. It takes two or more programs in the same (object-oriented) language and integrates or composes them according to some composition rule. HyperJ's pattern-matching abilities and bracketing (integrating a method body before, after or around another method body) are reminiscent of similar support in AspectJ's pointcut and advice. Nevertheless, HyperJ uses these features to integrate programs based on their structural information, due to its static join point model. AspectJ with its dynamic join point model, uses these features to identify points in the execution of a program, interrupt the program at these points and provide additional or alternative program execution. As such, AspectJ has powerful support for accessing values in the interrupted execution context and for manipulating these values in the advice, which is essentially a program in a different programming language.

We feel that despite their differences, or maybe because of them, composition (HyperJ) and pointcut/advice (AspectJ) approaches are complementary and a combination can provide powerful support for integrating programs. Intuitively, we can say that composition allows one to define an overall default strategy for composing two programs automatically, based on a mapping between program or execution elements. Aspects, on the other hand,

allow very selective and customised integration of programs, by specifying a set of join points and defining advice that is to be weaved at these join points. However, HyperJ and AspectJ are not compatible since they employ different join point models, static and dynamic, respectively. Later on in this dissertation, we argue that composition with a dynamic model is compatible with a pointcut/advice approach with a dynamic model.

Adaptation of rules or core functionality in an integration is an issues that can be addressed by all three approaches. However, in HyperJ, the adaptation needs to be expressed in a separate hyperslice. This requires an extra composition of this adaptation hyperslice either with the core functionality hyperslice or with the rules hyperslice. In AspectJ and JAsCo on the other hand, the adaptation is included in the encapsulated integration.

Since JAsCo is partially based on AspectJ it deals with the dependency issue in a similar way (not all shown in this chapter). The added value of JAsCo is its ability to express all kinds of customised combinations of rules, which is why we paid particular attention to this issue. JAsCo has by far the most sophisticated approach to combining rules. Compared to AspectJ, this is due to JAsCo's ability to control instantiation, initialisation and execution of aspects, which is exactly the major shortcoming we encountered in AspectJ.

Still, we observe that manually expressing all combinations of rules is a cumbersome endeavour. It becomes even more challenging when rules are frequently added to a set of rules that are activated together, as is often the case. Although the combination strategies are somewhat robust, they have to be adapted more often than not when a new rule is added.

In the following chapter, an alternative technology is introduced for expressing rule-based knowledge. *Rules* are modules in *rule-based languages* conceived especially for representing rule-based knowledge. Using a rule-based language is particularly useful for dealing with the combination of rules. As such, this issue is pushed to the rule-based knowledge representation and does not have to be dealt with in the integration.

Table 3.1: An overview of the integration issues and how they are addressed by the discussed aspect-oriented programming approaches.

	HyperJ	AspectJ	JAsCo
dependency	±	+	+
adaptation	±	+	+
combining rules	−	−	+

Finally, note that the description of JAsCo includes a part on its ability to reconfigure aspects at run-time. This, of course, has direct repercussions on the ability to plug and unplug rules at run-time. Since most approaches that attempt to make rules explicit in object-oriented software applications indicate that this is an important feature, we deemed it necessary to mention it here.

To conclude this chapter, we give an overview of the integration issues and how well the reviewed aspect-oriented programming approaches address them in Tab. 3.1. Note that the rather categorical "−" does not mean that a particular approach cannot express a particular issue at all. It merely means that this approach does not offer support for expressing this issue in a significantly better way than a standard object-oriented programming language does.

Chapter 4

Rule-Based Languages

In the previous chapters we have shown how we can make do with object-oriented programming techniques for representing rules explicitly and modularly as rule objects. However, elaborate constructions are required for managing the dependencies between rules and combining rules. In rule-based languages rules are expressed in their natural form and a rule engine takes care of determining which rules are applicable, managing dependencies between rules, and ordering and excluding rules.

This chapter presents the state of the art in rule-based languages. First, we define rule-based languages and review their characteristics and advantages (section 4.1). After this, we discuss backward and forward chaining (section 4.2). Two formalisms on which all rule-based languages are based are presented next, which leads to a section on logic-based languages (section 4.3) and a section on languages based on production rules (section 4.4). In each of these sections, a representative and successful concrete language is presented. We end this chapter with an in-depth discussion on the merits of the different chaining modes and formalisms.

The information in this chapter is for the most part based on Jackson's book *Introduction to Expert Systems* [Jackson, 1986] and Russel and Norvig's book *Artificial Intelligence, a Modern Approach* [Russel & Norvig, 1995].

4.1 Overview

Rule-based reasoning emulates human thought and problem solving based on the explicit representation of human knowledge as rules. A rule usually has the structure *conclusion IF condition* or a variation thereof. Consider again some of the example rules introduced in an earlier chapter:

- **LoyalDiscountRule:** *a customer gets a 5% discount if he or she is loyal*
- **LoyalCustomerRule:** *a customer is loyal if he or she has a charge card and good credit rating*

Although a rule resembles an if-then-else statement from procedural languages, it is not activated in any predetermined order relative to other if-then-else statements. On the contrary, rules are relatively independent modules and the rule-based language itself is responsible for determining the set of applicable rules. This eliminates the problems of implementing rules in procedural languages [Jackson, 1986]:

1. manually ordering and merging the conditional statements into one application flow,

2. loss of identity of the rules and

3. dealing with the impact of changing one single rule.

Although rule objects, introduced earlier in Chap. 2, deal with the second problem, it does not eliminate the other two.

Typically, a rule-based language is employed for representing rules. It consists of an *inference engine* or *rule engine* which automatically controls the selection and activation of rules. The engine activates a rule when incoming data matches the activation pattern of either its condition or its conclusion. As such, the engine automatically orders and merges the rules.

An inference engine can use the two example rules for inferring customers that get a 5% discount based on knowledge of their charge card and credit rating. Hence, dependencies between rules are also automatically taken care of because a rule engine *chains* rules together. There are two different rule chaining modes, backward and forward, which match data to the rule's conclusion or condition, respectively. Those two chaining modes are discussed in detail in section 4.2.

We define *rule-based languages* as systems or languages that provide

1. *rules*, modular structures for expressing rule-based knowledge,

2. data structures that are manipulated by active (or triggered) rules, and

3. an engine or interpreter for selecting and activating rules.

Note that this is a restricted version of the definition for *pattern-directed inference systems* [Jackson, 1986].

There exist two main formalisms for expressing rules: *production rules* and *first-order predicate logic*. Systems based on the former formalism are usually referred to as *production systems* and express rule-based knowledge as *production rules*, whereas systems based on the latter are often called *logic-based systems* where rules are *logic clauses*. As mentioned earlier, we use the terms *rule-based language* and *rule* to denote the corresponding concepts in both formalisms in a general way.

Both formalisms have representative systems: the logic programming language *Prolog*[Flach, 1994] based on first-order predicate logic and the production system *OPS5* [Brownston et al., 1985]. The two formalisms and their representative systems are discussed later on in section 4.3 and section 4.4.

4.2 Rule Chaining

The engine activates a rule when incoming data matches the activation pattern of either its condition or its conclusion. The second approach is taken by *backward-chaining* engines: they attempt to prove a conclusion by finding rules that would conclude it and trying to establish their conditions. *Forward-chaining* engines activate a rule when data matches its condition and generate its conclusion.

Note that chaining is not the same as reasoning: backward reasoning attempts to prove a particular goal whereas forward reasoning infers new knowledge starting from the existing knowledge. In principle, either chaining modes are able to express both kinds of reasoning strategies for solving problems [Jackson, 1986]. However, it is obvious that using the corresponding chaining mode for a particular reasoning strategy is more expressive.

We explain the two chaining strategies conceptually in the remainder of this section.

4.2.1 Backward Chaining

Backward chaining is initiated when the rule-based language is *queried*, after which it will try to find rules that can conclude the query and attempt to establish the rules' conditions in turn. If the query is successful, it results in a set of bindings for its unbound variables. Backward chaining is goal-oriented and has the advantage of only activating rules that can potentially contribute to the goal.

For example, if we want to know the discount(s) a certain customer is entitled to, we perform a query *discount of ?percentage for Jos*, where *?percentage* is an *unbound variable* which represents the actual discount percentage being inferred. The sequence of backward chaining steps for this example are illustrated in figure 4.1.

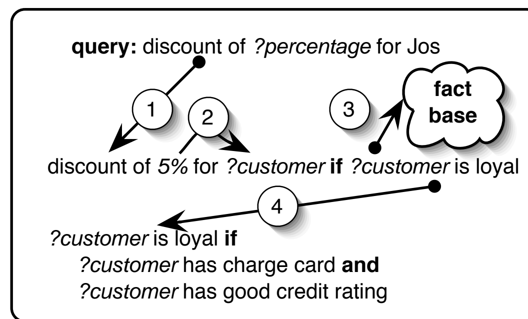


Figure 4.1: Backward chaining steps for inferring a discount percentage.

1. The inference engine looks for rules that conclude a discount percentage, such as the first rule in figure 4.1. The query is matched with this rule's conclusion, which binds the rule variable *?customer* to *Jos* and the unbound query variable (*?percentage*) to *5*.

2. Then the engine attempts to infer the condition of this rule, taking into account the current variable bindings.

3. First, the engine will look if there is a *fact* in the fact base with which the condition matches.

4. If not, it will again look for rules that can conclude the condition, such as the second example rule in figure 4.1. Again, the second rule's conclusion is matched with the condition of the first rule, taking into account the variable bindings.

The process is repeated: The engine attempts to infer both conditions of the second rule and so on. In the end, the query indicates failure, or success and solutions for the discount percentage. The example in figure 4.1 infers a *5%* discount for *Jos*.

Typically, analytic tasks such as classification, diagnosis and assessment are goal-oriented and solved by backward reasoning [Schreiber et al., 2000].

4.2.2 Forward Chaining

Forward chaining is initiated when a new fact is asserted. The rule-based language will try to find rules whose conditions match with the new fact, establish the other conditions, and assert the rules' conclusions as new facts.

For example, the fact *Jos has good credit rating* is asserted. The sequence of forward chaining steps for this example are illustrated in figure 4.2.

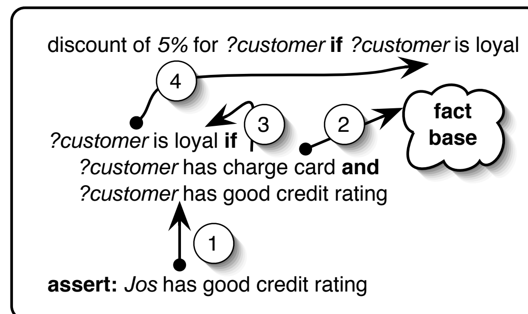


Figure 4.2: Forward chaining

1. The fact is matched with all conditions that mention credit rating for customers, in this case the second condition of the second rule in figure 4.2. This binds the rule variable *?customer* to *Jos*.

2. The engine attempts to establish the first condition of this rule, if *Jos* has a charge card, by looking for a fact which matches with it.

3. If such a fact is found, the rule's instantiated conclusion, *Jos is loyal*, is asserted as a new fact.

4. This assertion activates the forward chainer again, which matches that fact with the condition of the first rule in figure 4.2. As a result, the instantiated conclusion, *discount of 5% for Jos*, is also asserted.

In principle, this process can be repeated until all possible facts are inferred and asserted. However, there are rule-based systems that allow the forward-chaining inference process to be stopped before that happens: for example, an explicit halting statement might be provided, or the system stops automatically after a number of cycles or when a specific fact has been inferred.

Typically, synthetic tasks such as design, scheduling and assignment are usually solved by forward reasoning [Schreiber et al., 2000].

4.3 Logic-Based Systems

The origin of logic-based systems is first-order predicate logic. This logic is simplified and given a procedural interpretation in order to use it as a practical basis for logic programming. This section first presents first-order predicate logic, then logic programming, and finally the logic programming language Prolog.

4.3.1 First-Order Predicate Logic

Propositional logic is concerned with establishing the truth value of combinations of atomic propositions, which each have a definite truth value. The omnipresent examples of atomic

propositions are *Socrates is a man* and *Socrates is mortal.* These atomic propositions are not further analysed.

Predicate logic, on the other hand, exposes the internal structure of propositions and analyses them into predicates that are applied to individuals, which are objects the "world" consists of and have individual properties. Universal and existential quantification is permitted over the individuals.

Predicate logic can represent for example

- facts, such as *Socrates is mortal*
- general statements, such as the clause *all x (x is man → x is mortal)*, i.e. for every individual x, if x is a man then x is mortal
- vague statements, such as the clause *some x (x kills Socrates)*, i.e. there is some individual x, such that it kills Socrates
- complex relationships, such as the clause *all x, y (some z (z is a parent of x and z is a parent of y → x is a sibling of y))*, i.e. for all individuals x and y, there exists some individual z, such that if z is a parent of x and z is a parent of y then x and y are siblings

First-order predicate logic, which only quantifies over individuals such as *Socrates* and not over predicates such as *is mortal*, is complete and sound but not decidable.

4.3.2 Logic Programming

Logic programming is the use of logic as a programming language, which involves the mechanisation of inference. In this context logic is used for problem solving: we are interested in the inferences we can draw from a given set of proposition and more importantly, which ones solve a particular problem. The problem is that logic does not indicate the inferences that will most likely lead to a solution, but indicates all the inferences that are valid according to the inference rules. Hence, some built-in strategy or meta-rules are required to direct the selection of what to infer next. If not, more knowledge increases the search space since more inferences can be drawn, thus decreasing the likelihood of finding a solution.

In logic programming languages, the full syntactic variety of first-order predicate logic is usually avoided, instead using normalised syntactic schemes such as *full clausal form* or the *Horn clause subset.* The main rationale behind this is that less variety in the syntax reduces the number of inference rules required, which simplifies the mechanisation of inferencing. Logic programming languages that use clausal forms typically require only one inference rule, which we discuss later on.

All expressions from predicate logic can be transformed into full clausal form. In short, this is achieved by eliminating most of the connectors used in predicate logic as well as existential quantification. *Implication* is used as main connector, resulting in clauses of the form

$$b_1, ..., b_p \leftarrow a_1, ..., a_m, \neg a_{m+1}, ..., \neg a_n$$

where $p \geq 1$ and $n \geq 0$. We refer to $b_1, ..., b_p, a_1, ..., a_n$ as *atoms* and $\neg a_i$ is the negation of a_i. The a_i are implicitly conjoined conditions of the clause, whereas the b_i are implicitly disjoined conclusions.

If the clause contains variables, $x_1, ..., x_k$, they are implicitly universally quantified, interpreted as

for all $x_1, ..., x_k$ $(b_1 \vee ... \vee b_p \leftarrow a_1 \wedge .... \wedge a_m \wedge \ a_{m+1} \wedge ... \wedge \ a_n)$

Each atom is a predicate-argument expression, usually of the form

$p(t_1, ..., t_l)$

where t_i are terms and $l \geq 1$. A term can be a variable, a constant symbol or a function-argument expression.

The Horn clause subset is identical to the full clausal form except that there is only one atom at most on the left-hand side.

$p \leftarrow q_1, ..., q_m, \neg q_{m+1}, ..., \neg q_n$

Not all, but most expressions in predicate logic can be converted to Horn form.

The single inference rule required by logic programming languages that use clausal forms is the *resolution* inference rule. Consider the two clauses in full clausal form

$p_1 \vee ... \vee p_m \leftarrow q_1 \wedge ... \wedge q_n$
$r_1 \vee .... \vee r_k \leftarrow s_1 \wedge ... \wedge s_l$

where $q_1, ..., q_n, s_1, ..., s_l$ are atoms or negated atoms. If one of the atoms in the first clause's condition q_i with $1 \leq i \leq n$ matches one of the atoms in the second clause's conclusion r_j with $1 \leq j \leq k$ then we can infer the clause

$p_1 \vee ... \vee p_m \vee r_1 \vee ... \vee r_{j-1} \vee r_{j+1} \vee ... \vee r_k \leftarrow q_1 \wedge ... \wedge q_{i-1} \wedge q_{i+1} \wedge ... \wedge q_n \wedge s_1 \wedge ... \wedge s_l$

The basic pattern-matching operator in resolution, needed for matching atoms, is referred to as *unification*. Since atoms are predicate-argument expressions, only atoms with the same predicate and number of arguments can be unified. Two such atoms are unified by substituting their variables in a way that makes them identical. Unification should return the *most general unifier*, which is the substitution that makes the least commitment about the substitutions of the variables.

When we want to derive a clause P from a set of clauses S, it is desirable to adopt a goal-oriented strategy: we reason backward from the goal towards the evidence needed for proving the goal. *Resolution refutation*, also known as reductio ad absurdum or proof by contradiction, is a complete inference procedure which adopts the resolution inference rule but refutes the goal. As such, in order to prove that some clause P follows from S, we assume $\neg P$ and then attempt to derive a contradiction from its conjunction with S. If we succeed, we can assert P. In case of failure, we have not proved $\neg P$ but failed to prove P.

Suppose we attempt to resolve the negated atom $\neg q$ with the clauses in S, three possibilities arise:

- either there is no clause in S that contains q, hence the proof fails
- or S contains q (i.e. the clause containing only the atom q), hence the proof is immediate
- or S contains a clause $... \leftarrow ... \wedge q \wedge ...$, hence this clause resolves with $\neg q$ to generate

... $\leftarrow$ ..., whose atoms — interpreted as subgoals — all need to be resolved in order to prove the contradiction

However, although backward chaining tends to focus the search for a solution, it does not eliminate the combinatorial explosion inherent in the proof-generation process. A number of methods exist that attempt to deal with this, such as breadth-first search, which computes resolvents level per level. Although this method is complete, it is very inefficient.

The idea of logic programming is to express knowledge as a declarative program that specifies what the problem is instead of how it should be solved. The resolution inference rule embodies the procedural knowledge and is used to find a solution. Hence the procedural interpretation of a clause $A \leftarrow B_1, ..., B_n$ is *prove* A *by proving each of* B_1, *...,* B_n.

4.3.3 Prolog

Prolog [Flach, 1994] is by far the most widely used logic programming language. It uses resolution refutation on a knowledge base consisting of clauses in Horn form. Prolog uses the *linear resolution strategy*, always resolving with the (derived) goal, which results in linear proof trees. This method is complete for the Horn clause subset. Furthermore, Prolog has

- a built-in control strategy, more specifically depth-first search of the proof tree. Prolog *backtracks* when it fails until a solution is found or there are no more possibilities.
- a top-down search rule, more specifically selecting clauses nearer the top of the knowledge base first, which results in Prolog not being purely declarative
- a left-right computation rule, more specifically considering goal atoms or negated atoms from left to right

Prolog omits one expensive step in the unification algorithm, the *occur-check*, which is needed to detect, for example, that unifying `X` with `f(X)` should fail. Since the occur-check takes time linear to the size of the expression and is performed for each variable, it makes the time complexity of the unification algorithm $O(n^2)$. Although this omission makes Prolog unsound, actual errors happen very seldom in practice.

Since Prolog is a backward chainer, it is activated with a query, i.e. the initial goal of the inference process. Prolog queries can consist of conjunctions of atoms, negated atoms and variables.

A characteristic of Prolog is that it uses a *negation as failure* operator *not* instead of negated atoms in the condition of a clause. The interpretation of $not(p)$ is that it is considered proved if the system fails to prove p. Prolog also only allows assertion of $x = p$ where x is a variable.

Prolog has many built-in predicates for arithmetic, input/output and various system and knowledge base functions. Atoms using these predicates are proved by executing code rather than doing further inference.

An example Prolog program defines the *member* predicate:

```
member(X,[X|L]).
member(X,[Y|L]):-member(X,L).
```

Terms with uppercase characters denote variables. The square brackets `[` and `]` denote lists, where the expression before the vertical bar `|` is the first element of the list and the expression after is the rest of the list.

This predicate can answer different kinds of queries. It can be used to

- confirm that `member(2,[1,2,3])` is true
- enumerate the three values of `X` that make `member(X,[1,2,3])` true
- find the value of `X` such that `member(2,[1,X,3])` is true
- enumerate lists for which `member(1,List)` is true

The example rules presented earlier can be expressed in Prolog as follows:

```
discount(Customer,5) :- loyal(Customer).
loyal(Customer) :- chargeCard(Customer,Card),goodCredit(Customer).
```

Suppose the following facts are asserted:

```
chargeCard(jos,visa).
goodCredit(jos).
```

Again, the defined predicates can be used *multi-way*. A query `discount(jos,D)` can be performed, which in case of success results in the variable D being bound to the inferred discounts (one or many). Given the facts, this query will result in `D` being bound to `5`. Alternatively, a query `discount(jos,5)` can be performed to check if this is true or not.

In both cases, in order to prove the query, the engine attempts to establish the condition by finding and activating rules that can prove it. Specifically, in order to establish the condition with predicate `loyal`, the engine activates the last rule, which in turn requires the conditions with predicates `chargeCard` and `goodCredit` to be established. The engine first looks for existing facts that unify with these conditions, which is successful in the particular situation sketched here.

4.4 Production Systems

A *production system* consists of a set of production rules, a working memory that holds data, and a rule engine which employs some conflict resolution strategies. We discuss these three ingredients below and end this section with a discussion of OPS5.

4.4.1 Production Rules

Production rules present empirical associations between patterns of data and actions that should be performed as a consequence. As such they also have the format *conclusion* **if** *condition*, although the conclusion can sometimes be interpreted as recommending an action rather than concluding that a certain proposition is true.

The patterns and concrete data are usually represented as *object-attribute-value* triplets. The triplet represents concrete data if all the elements are values. It represents a *pattern* if the *object* or the *value* (or both) are variables. Patterns are matched against concrete data which results in the variables being bound. Note that the *objects* in the triplets are nothing more than constant symbols which denote a conceptual object and are not to be confused with objects which encapsulate data and behaviour in object-oriented programming languages.

4.4.2 Working Memory

The *working memory* holds concrete data in the form of the object-attribute-value triplets. The data is used by the rule engine to match to the rules' conditions. Two possibilities arise:

1. if one of the rule conditions has no variables, then it is satisfied only if an identical expression is present in the working memory
2. if one of the rule conditions has at least one variable, i.e. if it is a pattern, then it is satisfied only if there exists data in working memory which matches it, taking into account the rule's other conditions that have been matched

Pattern matching is performed as follows:

1. the first match, i.e. the first condition of a rule being matched, is simply an assignment of values to the condition's variables which would make the condition identical to the data it is matched against if the variables are substituted with the values
2. subsequent attempts to match, i.e. the other conditions of the rule, are constrained by the earlier substitutions

Pattern matching can be very expensive since all rules' conditions have to be matched against all the data in the working memory. A very efficient algorithm has been developed, the *RETE* algorithm, which deals with this many-to-many match [Forgy, 1982].

4.4.3 Conflict Resolution

The rule engine typically goes through the *recognise-act* cycle, which consists of the following steps:

1. matching: match conditions of rules against data in the working memory
2. conflict resolution: if there is more than one rule that could fire, collect them in a *conflict set* and decide which one to apply
3. application: apply the rule (which might result in adding or deleting data from the working memory), and go to step 1

The cycle stops if no more rules become active, more specifically if the conflict set is empty. The conflict set typically consists of *instantiations*, which are pairs of rules and variable bindings derived from pattern matching.

Conflict resolution strategies have been devised for dealing with non-determinism and focusing the program's attention on the problem at hand. Such strategies add to the heuristic power of production systems. A good conflict resolution strategy should exhibit sensitivity and stability. The former means responding quickly to context changes represented by the working memory. The latter refers to maintaining some kind of continuity in the line of reasoning. There exist several popular strategies, which are often used in combination:

1. refractoriness: a rule can only fire once on the same data
2. recency: instantiations of rules with more recent data are preferred
3. specificity: instantiations of more specific rules, i.e. rules with more conditions, are preferred

Whereas traditional production systems are usually implemented in Lisp, modern-day production systems are implemented in a state-of-the-art object-oriented programming language. The main difference with most traditional systems is that current systems use real objects as data in the working memory as opposed to the aforementioned triplets, which are also referred to as *pre-objects*[Pachet, 1992]. We elaborate later on in chapter on hybrid systems (Chap. 5).

4.4.4 OPS5

OPS5 [Brownston et al., 1985] is the most well-known production system language and the first to be based on the Rete algorithm. It was developed in the late '70 and early '80. OPS5 was used successfully for implementing a commercial system: a configurer of VAX computer systems, *R1* which was later renamed to *XCON*. The success of XCON led to the development of other substantial expert systems with OPS5.

A successor of OPS5 is *ART*, which improved and extended OPS5's expressiveness and reasoning capabilities. It was developed in the mid '80. *CLIPS*, the *C Language Integrated Production System* is an open source implementation of the forward chaining capabilities and syntax of ART.

More recent systems are *OPSJ*, the successor of OPS5, developed for Java by the *Production Systems Technologies, Inc.* with founder and chief scientist Charles Forgy, the inventor of the Rete algorithm. *Jess*, the *Java Expert System Shell*, is originally inspired by CLIPS but has grown into a distinct rule engine and scripting environment written entirely in Java. The common characteristic of this new generation of production system languages is that they use the objects of the underlying language directly instead of pre-objects, which are essentially represented in a datastructure in the underlying language, usually Lisp. Since these new systems integrate rule-based reasoning with object-oriented programming, they will be discussed in the next chapter on hybrid systems (Chap. 5).

In an OPS5 program, one first needs to declare the data structures that will be used by the rules. This is done using the `literalize` statements below, which essentially result in implementing these data structures as Lisp vectors. The first identifier after the `literalize` keyword denotes what kind of (pre-)object is being declared. The rest of the identifiers denote the attributes of the pre-object.

```
(literalize person name status card credit)
(literalize discount percentage)
```

In OPS5 production rules have the general form

```
(p <identifier>
  <condition>*
-->
  <conclusion>*
)
```

where the asterisks indicate one or more occurrences of condition or conclusion expressions.

The two example rules introduced in the beginning of this chapter can be expressed in OPS5 as follows:

```
(p LoyalDiscountRule
  (person ^name <name> ^status loyal ^card <c> ^credit <r>)
  (discount ^person <name> ^percentage <p>)
```

```
-->
  (modify <discount> ^percentage 5)
)

(p LoyalCustomerRule
  (person ^name <name> ^status <status> ^card {<c> <> nil} ^credit good)
-->
  (modify <person> ^status loyal)
)
```

4.5 Discussion

First, let us note that we did not consider advanced issues in rule-based languages, such as meta-rules for controlling inference, optimisation techniques and so on. We focus on the interface of rule-based languages, i.e. how they are activated, and only on the basic inference steps of the engines.

Although logic programming languages are not conceived with rule-based knowledge representation in mind, it is possible to use clauses for representing rules and facts for representing data of the working memory. Prolog also provides meta-predicates that correspond to asserting and removing data (facts) in the working memory. Unification in logic programming languages is more powerful than pattern matching in production systems. Moreover, backward chaining has some advantages: the inferences drawn are at least potentially relevant to the goal, expressed as a query, thus focusing the search.

However, despite its expressiveness and powerful mechanisms, Prolog has some disadvantages. First of all, representing defaults and exceptions requires some extra programming. Moreover, forward chaining is typically simulated by exploiting depth-first search and backtracking in Prolog. However, both unification and backtracking are computationally expensive in terms of both memory and processing power. Prolog does not have an equivalent of the Rete algorithm. Finally, there is some doubt with respect to the practical use of Prolog for developing larger-sized rule-based programs.

Production systems are not as powerful as logic-based languages, mostly because unification allows both the data and the pattern to have variables. OPS5 it is suitable for expressing empirical knowledge such as which actions to take in certain situations, defaults and exceptions. Moreover, some specific configuration tasks require a forward-chaining engine such as OPS5 provides and would be hopelessly inefficient in Prolog.

We conclude that rule-based languages are without a doubt more suitable for expressing rule-based knowledge than object-oriented programming languages. Nevertheless, the amount of rule-based knowledge should warrant the use of a rule-based language.

Rule-based languages are not a magical solution: it is very unlikely that well-defined rules at a conceptual level can be mapped directly to rules in a rule-based language. There is a trade-off between higher-level languages, sometimes referred to as expert system shells, which provide abstractions specific for a particular task, and more general-purpose languages such as Prolog. The diversity of available systems and languages, in part due to the different formalisms and different levels of abstraction, makes selection of a language a very challenging endeavour. Our preference goes out to the more general-purpose languages, since they can be used to implement any abstractions that are deemed necessary.

Some would say that Prolog is the ultimate general-purpose language for expressing human knowledge and simulating human problem solving. Still, even Prolog is not suitable for representing all problems, as mentioned earlier. Additionally, OPS5 is definitely also a programming language, although one could argue that it has a slightly higher level of abstraction than Prolog.

The flip side of general-purpose rule-based languages, is that they are essentially *programming* languages. Although more declarative than procedural and object-oriented languages, they still require programming skills and in some cases intimate knowledge of the semantics, especially with respect to control.

We find that Prolog and OPS5 are the ideal representative pair of rule-based languages because they are complementary in every respect. First of all, each is an exemplary ambassador of a rule-based formalism, Prolog for logic-based systems and OPS5 for production systems, since they are both well-known and widely used, and have proven their worth. Furthermore, they each employ a different chaining mode: Prolog uses backward chaining whereas OPS5 uses forward chaining.

Chapter 5

Hybrid Systems

The first chapter focuses on one technology for representing rule-based knowledge and object-oriented functionality explicitly and separately: object-oriented programming. In this context, our first concern is *program integration*. The second chapter adds a technology, aspect-oriented programming, for facilitating this program integration. The third chapter, however, increases the technological complexity of program integration considerably, since another technology is introduced: rule-based representation. This results in *different* languages being used for representing rule-based knowledge on the one hand and implementing object-oriented functionality on the other. Thus, our first concern shifts to *language integration* rather than program integration. Indeed, when the goal is to integrate programs represented in different languages, one first has to consider how the languages are integrated. This is especially true if the languages are of a different paradigm. In this chapter we review *hybrid systems*, which integrate a rule-based language and an object-oriented language. In the next chapters, we again consider integration of programs, implemented in different languages, based on aspect-oriented programming.

Numerous *hybrid systems* have been proposed over the years. We have conducted a survey of over 15 of such hybrid systems in order to investigate the level of integration of the two languages. To do so, we first identify the relevant language integration issues. We reported on the language integration issues and the survey in [D'Hondt et al., 2004].

We observe that the large number of existing hybrid systems indicates that integrating the object-oriented and the rule-based programming paradigms is a useful endeavour. More importantly, we note that the rule-based languages of choice to be integrated are either based on Prolog or on OPS5 — with a few exceptions.

This chapter first discusses several categories of systems that might be considered hybrid, but are not included in our survey (section 5.1). The next section introduces the language integration issues we identified (section 5.2). The following section reviews the results of our survey of hybrid systems with respect to these language integration issues: for each issue, different approaches are identified and discussed (section 5.3). Finally, we give an overview and conclusion of the survey (section 5.4). Note that a complete list of references to material on the surveyed systems — usually manuals or web sites — is included at the end of this chapter rather than in the bibliography.

5.1 Ruled-Out Systems

In our quest for existing hybrid systems we came across several kinds of systems that seem suitable at first glance but are discarded for different reasons. We review those here. Note

that the titles of the following subsections name the systems that are ruled out, not the ones we wish to include in our survey.

5.1.1 Frame-Based Languages

The object-oriented paradigm became increasingly popular both in the field of software engineering and in the field of artificial intelligence. These fields emphasise a different point of view about the notion of an object, respectively [Masini et al., 1991]:

- from a structural point of view, an object defines a data structure and a set of operations applicable to this structure
- from a conceptual point of view, an object represents a knowledge concept

Usually, object-oriented languages for software engineering are class-based. The historical leaders of these family of languages are *Simula* and *Smalltalk-80*. The latter is often considered as the archetype of class-based object-oriented programming languages because of its uniform model: the only data entity is the object and the only control structure is message sending.

Object-oriented languages in the field of artificial intelligence on the other hand, are usually referred to as *frame-based languages*. They are derived from network-based formalisms and Minsky's frame idea [Minsky, 1975]. They are usually prototype-based, where a frame describes a standard or typical situation or object. A frame holds a number of slots, each described by a number of facets, the most trivial being the slot's value. Contrary to objects from software engineering, a frame has no behaviour described by methods but local operations held by the slots which are activated on slot access. Typically, these are procedures written in the underlying programming language, such as Lisp as in, for example, *KL-ONE* and *KRL*.

Nowadays, the state-of-the-art object-oriented programming languages for software engineering are still class-based. Therefore, we only consider these kind of languages. Examples are C++, Java and Smalltalk.

5.1.2 Rule-Based Knowledge Attached to Frames or Objects

The first hybrid systems surface in the field of artificial intelligence and combine frames and rules. They were developed to address the difficulty of representing knowledge that is naturally expressed as rules, in an object-oriented style and vice versa [Jackson, 1986] [Fikes & Kehler, 1985]. These hybrid systems typically provide declarative as well as procedural facets to be associated with a slot in a frame. Declarative facets attach collections of rules to a slot which are activated when the slot is accessed. *KEE* [Hendrix, 1979], *LOOPS* [Stefik et al., 1983] and *CENTAUR* [Aikins, 1984] are the most notable hybrid systems of this kind.

Currently, there is a new surge of hybrid systems based on state-of-the-art (class-based) object-oriented programming languages: C++, Java, and Smalltalk. These "new generation" hybrid systems are accompanied by guidelines and methodologies for developing object-oriented software applications with explicit (business) rules [von Halle, 2001] [Ross, 2003] [Date, 2000]. The methodologies target these new hybrid systems for expressing the resulting software application in. They strongly advocate separating rule-based knowledge and object-oriented functionality in order to facilitate independent evolution and development. As such, the new hybrid systems differ from the first hybrid systems in that rules are not attached to frames (or objects) and activated when frame slots are accessed. Conversely,

rules and objects are defined separately, instead of rules being defined in the classes. It will be these new kind of hybrid systems that we consider. This excludes systems such as Aion from the survey, although we still mention Aion in the course of this chapter because some of its features are instructive.

5.1.3 Object-Oriented Extensions

We have already established that, for our purposes at least, a hybrid system should consist of a state-of-the-art, class-based object-oriented programming language. Let us restrict this further and say that a hybrid system should provide *native* object-oriented support. This contrasts with hybrid systems which consist of a language, not of the object-oriented programming paradigm, which emulates object-oriented programming. For example, encapsulation of state and behaviour, polymorphism and inheritance are some object-oriented characteristics that are easily emulated in Prolog. Hence, there exists a large number of object-oriented extensions of Prolog or other logic programming languages. Typically, a preprocessor translates an "object-oriented" logic program to standard Prolog or any other objectless logic programming flavour. Since this is clearly not a case of native object-oriented support, we eliminate Prolog systems such as LogTalk, Prolog++, Trinc Prolog, NUOO-Prolog, ALS's ObjectPro, OL(P) and IF/Computer's MINERVA from our survey.

In addition to the Prolog-based systems mentioned above, a number of systems exist that are based on *linear logic*. Without going into detail, this is an extension of classical logic with features such as a declarative notion of state and the ability to express problems involving concurrency. Apparently, higher-order linear logic is a suitable medium for defining the semantics of object-oriented languages, such as messages, inheritance, dynamic dispatch, object update and object extension [Bugliesi et al., 2000]. As such, a number of logic language based on linear logic have been proposed: Lolli, Lygon, PRObjLOG and YahOO. Needless to say that these language are not taken into account, although they propose an integration of object-oriented and logic programming as well.

5.1.4 Translators

Some systems implement a rule-based language in an object-oriented language and provide a translator for translating the former into the latter, or in the language understood by the virtual machine of the latter. However, such systems do not provide an integration of the two languages from a programming point of view. The programmer has to select one of the languages for expressing his or her solution in. There is no support for expressing parts of the solution in the most suitable paradigm and integrating the parts to form one fully functional program. Examples of such systems are jProlog, KLIJava and NetProlog.

5.1.5 One-Way Hybrid Systems

Many hybrid systems that combine an object-oriented and a rule-based language are constructed by implementing the latter in the former. While this is an excellent set up for providing a means for manipulating values and activating behaviour from the object-oriented language, there are some systems that do not exploit this. These systems simply use the object-oriented language as an implementation language without providing an interface to activate it. This allows them to embed the rule-based language in a graphical user interface or web interface, implemented in the object-oriented language. Hence, these systems provide an API for activating the rule-based language from the object-oriented language.

We refer to these systems as one-way hybrid systems and discard them from our survey. Examples are jProlog, W-Prolog, XProlog, IF/Prolog and LL.

5.1.6 Rule Management Tools

Rule management tools do not provide a dedicated language for expressing rule-based knowledge but offer some support for using object-oriented programming itself to represent rules. Typically a library is provided for loading rules, stored in XML for example, and activating them from the core application. The rules themselves are written in object-oriented programming as if-then actions. Thus features of logic reasoning systems are missing, like automatic control of the order in which the rules are fired and rule chaining. Although these tools are useful for "single-shot" rules, they are not powerful enough for expressing rules that depend on more involved decisions with many conditions and many rules [von Halle, 2001]. Examples of such tools are Business Rule Beans and OptimalJ.

5.1.7 Data-Change-Oriented Systems

Another kind of system that is not taken into account is data-change-oriented rule systems [von Halle, 2001]. These systems provide an automatic mechanism for activating rules when monitored data in objects changes. But there is no rule engine which manages the control flow of rules or provides conflict resolution. Some of these systems monitor database rows directly that represent business objects, such as Versata and USoft. Another kind of data-change-oriented system is integrated with an object-oriented language and monitors the instance variables of the objects, like R++. When rules are not bound to a particular object or data but are "free-floating", when many conditions have to be assessed or when the task is rule-intensive, a data-oriented rule system is not suitable [von Halle, 2001].

5.2 Language Integration Issues

Object-oriented functionality and rule-based knowledge should be integrated in order to obtain one correctly functioning software application. Since at this point we are integrating programs implemented in different languages, we want to investigate the integration issues at the language level. To do so, we'll first take a look at an example of how functionality implemented in an object-oriented language and knowledge represented in a rule-based language interact with each other (section 5.2.1). Then we list the language integration issues and discuss them in a conceptual way, considering the general characteristics of object-oriented and rule-based languages (section 5.2.2 to section 5.2.7).

5.2.1 Example

In figure 5.1 the example e-commerce functionality and rules are reviewed again. Object-oriented functionality (left) and rule-based knowledge (right) are separate, independent and meaningful programs.

Associations between objects are denoted by arrows connecting the white circles (objects). Hence, figure 5.1 shows snapshots of the execution of the object-oriented functionality. There are objects denoting a *customer*, an *order*, a *shopping cart* and three *products*. The bottom snapshot is when a customer has added products to the shopping cart. The top snapshot is when an order has been created for a customer of his or her products.

Rules have the form *conclusion* **if** $condition_1$ **and** ... **and** $condition_n$. Conclusions and conditions are bounded by boxes with rounded corners. The example backward-chaining

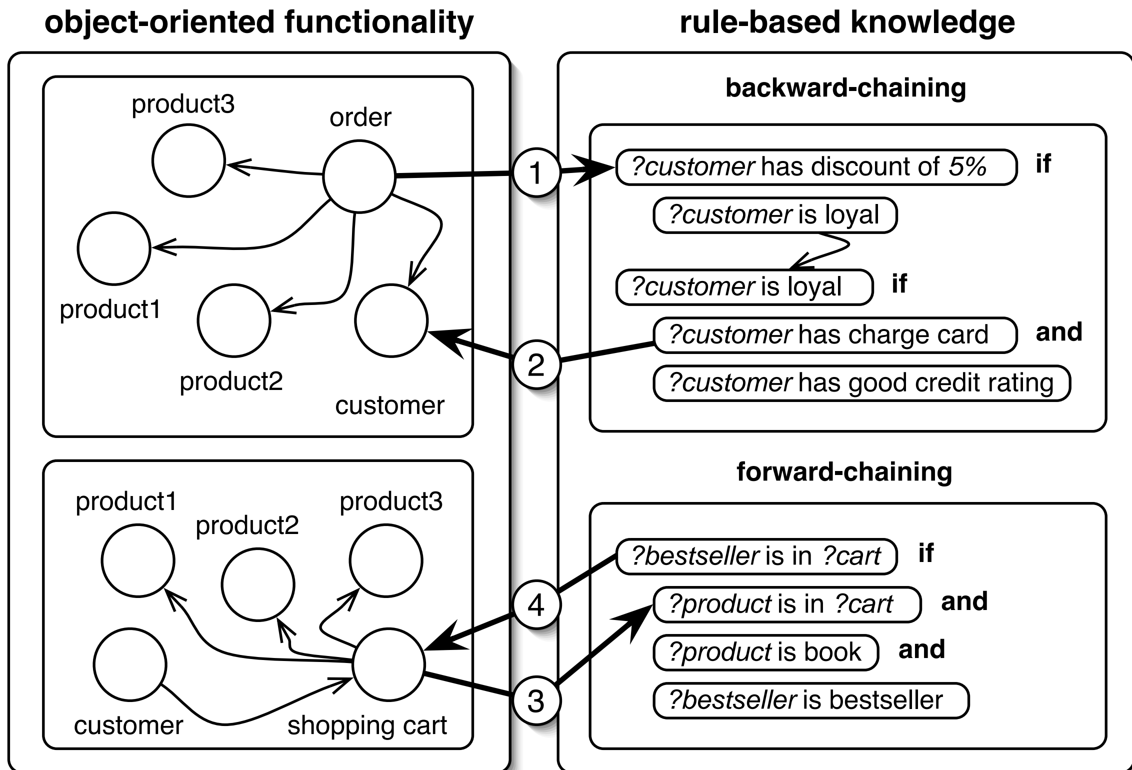


Figure 5.1: Example interaction between object-oriented functionality and rule-based knowledge.

rules can be activated by a query about a discount. The condition *customer is loyal* in the first rule is proved by activating the second rule. The forward-chaining rule is activated with asserting a fact about a product being in a cart. Note that many more similar backward-chaining and forward-chaining rules may exist and be activated by the same query and assert, respectively. Moreover, more rules can be chained in these inference processes. However, we only show these few rules for the sake of simplicity. A last remark concerning the rule-based part of figure 5.1 is that it shows rule definitions rather than a snapshot of the execution of the rules. When we review the integration issues in this example below, we describe the execution of the rules.

The thicker black arrows in figure 5.1 denote the integration between the two programs. Numbers 1 and 3 concern the activation of rule-based knowledge from object-oriented functionality, whereas numbers 2 and 4 denote the opposite. We review them here, taking their numbering into account:

1. An order object performs a query in a method to determine the discount when calculating the total price of the purchased products. This means that certain objects are passed when the query is activated. In particular, when the first rule is activated for proving the query, the rule variable customer has to be bound to the customer object. Furthermore, the activating method in the order expects a result, an inferred discount percentage, which has to be returned when the inferencing is done.

2. While inferencing, the condition *customer has good credit rating* has to be established. The rule engine sends a message to the values bound to the customer variable in order

to achieve this. Again values are passed when sending a message. Moreover, the rule engine expects an indication of success or failure as a result.

3. When adding a product to a shopping cart, an assertion is made that the product is in the cart. Again, these particular objects are passed with the assert. These objects are matched to the first condition of the rule, which binds the variables product and cart to the corresponding objects. Since forward-chaining rules have side effects as inference results, they do not “return” values to which rule variables are bound.

4. When the rule engine concludes that something is in a cart, it sends a message. Again, receiver and parameters are values to which rule variables are bound. The forward-chaining engine does not expect any result.

This example shows that rule-based knowledge — whether forward or backward-chaining — is activated more or less in the same manner. The same can be said about activating object-oriented functionality from the rules: forward and backward-chaining rules perform this similarly. The example also indicates that an activation typically consists of three steps: the activation itself, passing values, and returning results. This results in six language integration issues

1. activating rules in objects
2. returning inference results
3. objects as rule values
4. sending messages in rules
5. returning message send results
6. rule values in methods

These are discussed in a more precise and elaborate way in the remainder of this section.

5.2.2 Activating Rules in Objects

Typically, each language has to be outfitted with mechanisms, hidden or not, for activating behaviour in the other language. In languages of the same paradigm the integration is facilitated by the similarity of the means for activating behaviour. A possible approach is to adapt the evaluator of each language to recognise behaviour activation statements that should be delegated to the other language. Usually, some syntactical mapping allows the conversion of such a behaviour activating statement to the equivalent statement in the other language. Such a scheme is presented in the work on reflective extensions of open programming languages [Steyaert, 1994] [De Meuter, 1998]. In these approaches both languages are object-oriented.

In our work, the languages are of a different programming paradigm, which means they do not map so easily on a semantic level in addition to the syntactic differences. The problem here is that object-oriented languages in their simplest form basically have one means for activating behaviour, and that is by sending messages. In the context of a hybrid system, an object-oriented language also needs to support the activation of behaviour in the rule-based language. Hence, these two means for activating behaviour have to be consolidated.

5.2.3 Returning Inference Results

When a message is sent in an object-oriented program, a value which is the result of evaluating the corresponding method is returned, even if it is a null value. Hence, when an object-oriented program activates a rule-based program, the result of evaluating the rule-based program — the inference results — also have to be returned. However, inference results of forward-chaining rules are side-effects and cannot be returned. A backward chainer, however, binds values to rule variables as a result of inferencing. Hence, the approach to "returning" inference results has to be consolidated with the one to returning message send results. Another discrepancy is that rule-based languages manipulate single or multiple inference results in the same way, whereas object-oriented programs expect a single result.

5.2.4 Objects as Rule Values

When rule-based reasoning is activated from an object, as is illustrated by arrows 1 and 3 in figure 5.1, objects are passed. For example, when performing a query for obtaining the discount in an order object, the customer who places the order has to be passed to the discount rules. The rules need the customer for determining if he or she is loyal in this example. Likewise, the product which is added to the shopping cart is a part of the activation of the recommendation rules from the shopping cart. The recommendation rules determine what has to be added to the shopping cart based on this product object. Therefore, the rule engine needs to be able to bind rule variables to objects. Moreover, the pattern matcher or unification algorithm need to be adapted in order to match or unify over objects.

5.2.5 Sending Messages in Rules

Arrows 2 and 4 in figure 5.1 illustrate that rule-based knowledge invokes messages on objects. The statements in the rule-based language that activate behaviour are typically concluding a new fact in forward chaining, and establishing a condition in both backward and forward chaining. Note that the latter is achieved differently by both engines. Again, the challenge is integrating message invocation one way or another in the rule-based language.

Another pitfall is that a condition which needs to be established can contain unbound variables which are matched to concrete data by unification or pattern matching. Invoking a message with values which are interpreted to be valueless is not meaningful, and has to be considered when integrating the languages.

5.2.6 Returning Message Send Results

The rule engine expects an indication of failure or success when it attempts to establish a condition. Hence, when a message is sent in order to establish a condition, it has to be interpreted as returning success or failure. Moreover, message sends might return objects and not just boolean values (which are also objects in pure object-oriented languages). Therefore, the rule-based language has to foresee a mechanism for binding a message send result to a rule variable in order to use it while inferencing. Again, there has to be a way to indicate whether a collection of objects is a single binding or multiple bindings for a certain rule variable.

5.2.7 Rule Values in Methods

Rule variables are bound to rule values while inferencing. When a message is sent in a rule, these rule values are passed to the object-oriented language. Moreover, a special kind of rule value is an unbound rule variable. Since an active rule can contain unbound rule variables, these variables can also become parameters in a message send. Additionally, when the object-oriented program wants to activate an unbound query in order to employ the inferencing capabilities fully, unbound rule variables need to be created ex nihilo.

5.3 Survey of Hybrid Systems

We considered close to 50 systems that combine object-oriented and rule-based language features in one way or another. After eliminating unsuitable systems discussed earlier in section 5.1, more than 15 remain which we examined more thoroughly. This section presents the results of our survey of these systems with respect to the language integration issues presented in section 5.2. For each of these issues, we generalise, discuss and illustrate the different approaches we identified. Note that the examples are just snippets of code, often simplified, but representative of the particular approach they illustrate. But first, we address the typical implementation approaches of the surveyed systems.

5.3.1 Implementation Approaches

Symbiotic Systems

A first implementation approach is taking features of both languages and design one language that encompasses all of them. Typically, this is achieved by starting from an existing language and extending it with features from the other language by changing the evaluator of the language. In that case, there is one evaluator which is able to interpret behaviour activating statements of “both languages”. Furthermore, such a hybrid language typically contains the union of values of both languages, which can be used in both “languages”. Examples of systems that take this approach are Kiev, which extends Java with logic programming and Aion which provides object-oriented and rule-based programming. We refer to this category of hybrid systems as *symbiotic systems*.

Symmetric Systems

A second approach is the integration of two languages with two distinct implementation platforms. Examples are K-Prolog and its Java Interface to Prolog (JIPL), and SICStus and its bi-directional interface Jasper. Both systems provide Prolog implementations in C and an interface between programs written in Java and programs written in Prolog. In both cases, the Java-side of the interface consists of a library representing the Prolog system which uses the Java Native Interface to call C functions from Java. In SICStus, the Prolog part of the interface uses Java as a foreign resource, which requires a mapping between Java and Prolog, defined manually in external files. In K-Prolog, this consists of an API in Prolog for calling Java. We refer to this category of hybrid systems as *symmetric systems*.

A system which has another implementation approach but provides support for integration which is similar to that of SICStus is CommonRules. Although it is an implementation of a logic-based systems in Java, the developer also needs to define mapping files manually for integrating the logic language with Java.

Library Systems

The third and most common approach is to start from one language implement the other language in it. In this case, the former is referred to as the base language and we call the second the implemented language. If the implemented language anticipates integration with the base language, this typically ensures a good integration in one direction: the implemented language has built-in support for activating behaviour in the base language and implemented values are easily converted in base values. If not, the implemented language is not able to activate the base language at all. However, usually there is no such built-in support in the other direction: the base language is able to call the implemented language via an API. This is because implementing one language in a base language is one thing, but changing the evaluator of the existing base language in order to provide built-in support for activating the implemented language is another.

Since we only consider hybrid systems which at least consist of a state-of-the-art object-oriented programming language, the systems which follow this third implementation approach use the object-oriented language as the base language and implement a rule-based system in it. Examples of such systems which provide an API in the object-oriented language to activate the rule-based language and foresee built-in support in the latter for activating the former are OPSJ, JRules, NéOpus, Eclipse, Blaze Advisor, QuickRules, Jess, SOUL, K-Prolog, B-Prolog, Prolog Cafe, Jinni, InterProlog and XProlog. There is one system, JavaLog, which, in addition to the API, provides a special construct in the object-oriented language for calling the rule-based language. Hybrid systems that follow the third implementation approach but did not anticipate calling the object-oriented language from the rule-based system are jProlog, W-Prolog and IF/Prolog. We refer to this category of hybrid systems as *library systems*.

We conclude from our discussion on implementation approaches that most hybrid systems consist of an implementation of a rule-based language in a state-of-the-art object-oriented programming language. A recurring approach is to design the rule-based language in such a way that it is able to deal with object-like values in the rules instead of just pre-objects (in traditional OPS5-like languages) or literals (in traditional Prolog-like languages). Here as well there are different levels of integration, which are discussed in detail later on in the remainder of this section.

5.3.2 Activating Rules in Objects

We discern three approaches to activating the rule-based system from an object-oriented program.

Activation via API

The first kind of approach is employed by most library systems and symmetric systems, which provide a well-defined API for activating the rule-based language. Typically, an object is created and optionally initialised which represents the rule-based language evaluator. Then an API method is invoked on this object for activating rule-based behaviour. For this approach, the example code shown below is a representative way of activating a production system from Java. Suppose we have a person object and an object that represents the rule engine, which are both initialised later on. At some point, the message `assert` is invoked on the engine object, which is the way to activate a production system. When the rule-based

language is Prolog or another predicate-based language, the engine is typically activated in a similar way with a query instead of assert.

```
Person jos = new Person();
RBSystem engine = new RBSystem();
...
engine.assert(jos);
...
```

A variation hereof is provided by NéOpus. Rules have names which can be used as message selectors in object-oriented code. Such message sends result in the corresponding rule being activated.

Activation in Rule-Based Environments

A different approach that hides more of how the rule-based language is activated, introduces a special environment in the object-oriented language. This is the approach taken by JavaLog and several older Lisp-based hybrid systems such as Aion and KnowledgeWorks. They allow the definition of a rule-based code snippet in a method body, bounded by delimiters. For example, the JavaLog code below is part of a normal method body. The statements between {% and %} are evaluated by the rule-based language. Note that objects are referenced in the rule-based statements, in this case using the special character #.

```
Person jos = new Person();
Person li = new Person().
...
{% brother(#jos#, #li#), partner(#jos#,P). %}
...
```

Transparent Activation

Finally, some hybrid systems support transparent invocation of rules. Kiev, which is a symbiotic system, extends Java with logic programming so a query activation statement looks like normal message send statements. Moreover, rules are defined similar to methods, as shown below.

```
public rule brother(pvar Person p1, pvar Person p2) {
...
}
```

Invoking a query in Kiev with the `brother` predicate in a method body looks like the following. Note that the query activation statement cannot be distinguished from a normal method invocation statement.

```
Person jos = new Person();
Person li = new Person().
...
brother(jos,li);
...
```

5.3.3 Returning Inference Results

In general, hybrid production systems “return” inference results implicitly. Conversely, backward-chaining hybrid systems, especially those that are based on logic, have to indicate failure or success in proving a query. In case of success, bindings for unbound query variables need to be returned to the object-oriented program that activated the query. Moreover, there is the challenge of dealing with multiple inference results.

Most logic-based hybrid systems activate a query via an API message send, which of course returns a result. The return value is used to indicate success or failure of proving the query, which translates very naturally to the corresponding boolean values of the object-oriented language. Also, in case of success, most systems represent the bindings of any unbound query variables by setting the inferred value in the object-oriented representation of the variables. When we discuss the issue concerned with rule values in methods later on, we explain how these variables are represented in the object-oriented program.

We discuss the four approaches to returning inference results to the object-oriented program that activated the inference process. First, there is the approach returning results implicitly of hybrid production systems. Then, there are the three approaches of the logic-based hybrid systems to dealing with possible multiple solutions of a query.

Implicit Result

Typically, forward-chaining hybrid systems, most notably the ones based on production rules, do not “return” any inference results explicitly. They infer new objects to be put in the working memory, or update existing objects. Hence we conclude that these inference results are returned implicitly.

Note that when object-oriented programs activate the rule-based language via an API call, often the result of that particular call is returned. For example, the result of sending `assert` to the object representing the engine might result in the asserted value being returned. This, however, is not necessarily the result of the inference process activated by the assertion.

Deterministic Result

First of all, deterministic querying results in only one of the bindings being considered as a solution to the query. If more than one binding is inferred, only the first is returned.

Collection Result

A second approach collects all inference results, converts it to a collection object and returns it to the client method.

Quickrules, although a hybrid production system, returns a list of changed objects as inference results because the rules manipulate copies of the application objects.

Streamed Result

A third approach streams the results. The client method is able to repeatedly retrieve them, usually with API calls, until there are no more. A typical way of doing this is shown below, where `var` represents a query variable which is repeatedly set to its next solution. As said before, more on representation of rule values and variables later.

```
RBSystem engine = new RBSystem();
...
while (engine.nextSolution())
{ System.out.println(var.toString()); }
...
```

Kiev, however, supports streaming of results with a `foreach` statement, as a built-in language feature.

5.3.4 Objects as Rule Values

Most hybrid systems provide some automatic, bi-directional conversion between literals in both languages. This means that integers, strings, booleans and the like usually do not require any manual treatment when being passed from the one language to the other. This is especially true for library systems where the literals in the rule-based language *are* the literals from the object-oriented language. Furthermore, some systems convert the basic collection data structures of each language automatically: from object-oriented arrays to rule-based lists and the other way around. When discussing the issue of activating the rule-based system from object-oriented programs, we already mentioned objects being introduced in the rule-based system.

Basically, we distinguish four different approaches: (1) only literals as rule values, (2) wrapped objects, (3) deconstructed objects, and (4) objects that are used directly in the rule-based language.

Literals as Rule Values

As mentioned before, there are some library systems which only provide facilities for activating their rule-based languages, but where these do not incorporate any features from their base object-oriented language. In this case, no matter how the rule-based language is activated, the only object-oriented values allowed to be passed to the rules are literals. These systems are jProlog, W-Prolog, XProlog and IF/Prolog.

Deconstructed Objects as Rule Values

CommonRules and InterProlog are examples of the deconstruction approach. They both convert objects passed to their rule-based languages into facts. The first system requires the manual specification of the mapping of objects of a certain class to facts with a certain predicate. For example, objects of class `Person` which has attributes `name` and `age` map onto a fact with predicate `person` and three arguments: the first holds a reference to the object, the second and third the values of the name and age attributes, respectively. The object reference can be used to look up the actual object in some kind of object registry.

InterProlog, on the other hand, employs serialisation of objects in Java for deconstructing them. Hence, only Java objects which implement the `Serializable` are passed to Prolog. Such objects are serialized to a stream of bytes with the Java Serialization API. This format is then converted to Prolog terms using the InterProlog Object Grammar. In fact, only objects which are manipulated in Prolog need to be deconstructed this way. Alternatively, Java objects are registered in the Prolog engine's object table after which their references are wrapped in an `InvisibleObject` instance and used as such by Prolog.

Wrapped Objects as Rule Values

The other approach for manipulating objects in the rule-based language is by wrapping them. Even if it seems as if objects are used directly in the rule-based language, they are usually wrapped. The transparency is achieved by automatic wrapping and unwrapping of the objects.

Some systems do not provide such transparent support, which is illustrated with the code below, shown earlier for explaining how objects are passed to the rule-based system.

```
Person jos = new Person();
ObjectTerm o1 = new ObjectTerm(jos);
```

The rule-based language has extra means for manipulating the (wrapped) objects. First of all, objects can be bound to rule variables. Furthermore, the unification or pattern matching algorithms are extended to take unification or matching of objects into account. Finally, there are some features specific to hybrid production systems. A first feature is an extension of the basic production rules with statements for accessing and mutating object state directly. A second feature is that the wrapped objects are used by the operators for manipulating working memory elements: update, delete and assert. This is illustrated below by sample code in OPSJ.

```
rule person_rule
if {
p isa Person with (
nationality of p == B,
age of p > 18);
} do {
p.delete();}
```

Objects as Rule Values

Kiev, being a symbiotic language, only uses objects as data structure, by both object-oriented and rule-based language features. JRules, on the other hand, is a library system but also states that objects are used directly by the rules without them being wrapped in another representation.

5.3.5 Sending Messages in Rules

We identify four approaches to sending messages in the rule-based system.

Message Send Statements in Rules

A first approach, provided by most library systems based on production rules, extends the traditional rule-based system with message send statements in conditions and conclusions, referred to as *condition message send statements* and *conclusion message send statements*, respectively. This is illustrated below with the example rule that sends a message to objects matched with the variable `p` to obtain their nationality. The message send statements are evaluated as part of the inference process, simply by executing them in the object-oriented language. They should evaluate to true or false to indicate success or failure of the "condition", or are used in combination with operator to bind the result to a rule variable. Systems that use this approach are OPSJ, JRules, Eclipse, QuickRules, Aion, KnowledgeWorks and Kiev.

```
rule person_rule
if {
p isa Person with (
p.getNationality == B);
} do {
...}
```

Hybrid production systems have an action part in their production rules. Traditionally, the action consists of statements for deleting, asserting or modifying working memory objects. The current hybrid production systems, such as JRules, Eclipse and QuickRules, augment this with standard object-oriented message passing statements, so that the modification of objects can be expressed in object-oriented programming style. The evaluation of these statements happens in the same way as the message passing statements in the conditions, except that they do not necessarily need to return a boolean or a value to be used in a comparison operator. OPSJ actions are even pure Java code, which uses an API for deleting and asserting objects in the working memory.

Predicate API for Sending Messages in Rules

A second approach was discovered which defines special predicates which form a rule-based API to invoke object-oriented behaviour. This is the approach followed by most library systems based on Prolog. These systems offer similar predicates to the ones from K-Prolog:

- `javaConstructor(Class(Args,...), Instance)`
- `javaMethod(ClassOrInstance, Method(Args,...), Return)`
- `javaGetField(ClassOrInstance, Field, Value)`
- `javaSetField(ClassOrInstance, Field, Value)`

Note that the first predicate is used to create objects *ex nihilo* in the rule-based language. The resulting object is handled similarly to an object passed explicitly to the rule-based language.

The predicates are "uni-way" as opposed to traditional Prolog predicates. The convention is that all variables but the last have to be bound when a term with such a predicate is evaluated. Below we show an example rule which uses the above predicates, based on an example from Prolog Cafe.

```
main :-
  java_constructor('java.awt.Frame', X),
  java_method(X, setSize(200,200), _),
  java_get_field('java.lang.Boolean', 'TRUE', True),
  java_method(X, setVisible(True), _).
```

Typically, the first argument is bound to a class name or the representation of an object in the rule-based language. If the rule-based language has to establish a term with such a predicate, the arguments are converted to their object-oriented counterparts and the message send is executed in the object-oriented language. The result is converted to rule-based representation and unified with the last argument of the term.

Systems that follow this approach are K-Prolog, B-Prolog, Prolog Cafe, JavaLog, Jinni, InterProlog , SICStus Prolog and the expert system Jess.

Activation in Object-Oriented Environment

A different approach that hides more of how the object-oriented language is activated, introduces a special environment in the rule-based language. This is the approach taken by SOUL. It allows the definition of an object-oriented code snippet in a rule, bounded by delimiters. Consider, for example, the SOUL rule below. The statements between [and] are evaluated by the object-oriented language. Rule variables can be used in the object-oriented code snippet. The snippet is then evaluated by the object-oriented language for each value to which the variable is bound. In the example, the variable `?customer` is used in the code snippet.

```
loyal(?customer) if [?customer hasChargeCard]
```

Mapping Predicates in Rules onto Methods

CommonRules implements a third way of invoking messages from their rule-based languages. They require programmer-defined mappings of predicates to methods: when the rule engine needs to establish these predicates it will do so by invoking the method instead of searching for a rule. In CommonRules, not just any method can be the target of the mapping, only methods in specific classes. Moreover, the method must take exactly one argument: the object-oriented representation of the term being mapped which needs to be taken apart by the method itself.

5.3.6 Returning Message Send Results

When the rule engine expects a result other than an indication of success or failure from a message send, it needs a mechanism to manipulate the result. All approaches bind the result to a rule variable, we distinguish two main approaches.

Explicit Message Send Results

This approach is taken by logic-based hybrid systems which provide an API for sending messages in rules. The API is a uni-way predicate, which means that the last variable has to remain unbound so that the message send result can be bound to it. Because there is an extra predicate argument foreseen, the message send results are explicit. Typically, message send results are treated as a single binding for a rule variable.

Implicit Message Send Results

Message send results are dealt with explicitly by hybrid production systems which extend the production rule language with message send statements. Typically they provide an operator which matches the result to a rule variable. For example, in JRules this is the operator `from`:

```
when {
  ?l: LoanApplication(loanAmount > 100000);
  ?b: Borrower from ?l.getApplicant(); }
then { ... }
```

This approach returns message send results much like standard object-oriented languages do, so we call it implicit.

Note that some approaches provide an alternative operator for binding a collection as multiple results to a rule variable. In JRules, for example, this is the operator `in`.

SOUL, which sends messages from object-oriented code snippets in rules, also takes an implicit approach to returning message send results. It uses the predicate `equals` for unifying a message send result with a rule variable. For example, if the variable `?c` is bound to a customer object, the following term unifies the customer's shopping cart, retrieved with the accessor `shoppingCart`, with the variable `?cart`.

```
equals(?cart, [?c shoppingCart]).
```

5.3.7 Rule Values in Methods

Predicate-based languages have *terms* as values, which typically consist of a predicate and its arguments. Arguments can be values — literals or even objects — or unbound variables. We discuss the different approaches to representing these rule values in the object-oriented program.

Rule Values as Strings

The first approach simply passes a string which represents the query when activating the rule-based language. This string is subsequently parsed by the rule-based language. Since the string should conform to the syntax for queries, it can include variable names. The code fragment below illustrates this. While parsing the string, the rule-based language recognises the rule variable `X`.

```
RBSystem engine = new RBSystem();
...
engine.query("member(X,[1,2,3]");
...
```

Rule Values as Term Objects

The second way of representing rule values is by constructing the predicate-based structure, referred to as a *term*, out of objects. Since systems that provide the API-based approach to activating the rule engine usually implement it in the object-oriented language, there exists a representation of rule-based data — terms — in objects. Typically, this results in a program similar to the one shown below. It shows a compound term object being passed to the engine. The compound term objects consists of a predicate term object denoting the name of the predicate and the number of arguments, and the term objects which are the arguments of the predicate. In the example, an object is passed to the rule-based language, which is wrapped in an object that denotes objects in the rule-based language, more specifically an object term. The other argument of the query being constructed is an unbound variable, represented by a variable term object.

```
Person jos = new Person();
RBSystem engine = new RBSystem();
...
Predicate pred = new Predicate("brother",2);
ObjectTerm o1 = new ObjectTerm(jos);
VariableTerm v = new VariableTerm();
CompoundTerm t = new CompoundTerm(pred,{o1,v});
engine.query(t);
...
```

Rule Values are Objects

Current hybrid production systems replace the traditional pre-objects with objects. Hence the working memory is populated with objects and the engine is activated by changing or asserting an object in the working memory[1]. Therefore, when activating such an engine in an object-oriented program via an `assert` API call, an object can be used directly as the argument. We re-examine the example used earlier for illustrating how to activate an engine via an API.

```
Person jos = new Person();
RBSystem engine = new RBSystem();
...
engine.assert(jos);
...
```

As mentioned earlier, production systems perform actions on working memory objects as a result of inferencing. Therefore we conclude that hybrid production systems never introduce rule values in the object-oriented program.

Of the logic-based hybrid systems, both JavaLog and K-Prolog also pass objects directly to the rule-based language after which they are converted automatically to their rule-based representation.

JavaLog provides an unquote mechanism in order to substitute objects in the predicate string. The objects are passed in an array as the second argument of the `query` message. Again, rule variables are simply part of the string.

```
Person jos = new Person();
Person li = new Person().
Person persons[] = {jos,li};
RBSystem engine = new RBSystem();
...
engine.query("brother($0,$1)",persons);
...
```

K-Prolog activates queries using the predicate name and an array of objects which represents the list of arguments. Rule variables are represented by the null object in the array.

Rule Variables as Parametric Types

The fourth approach is exhibited by symbiotic systems, such as Kiev, which extend the object-oriented language with logic programming features. The code fragment below shows the definition of a rule with predicate `brother` and two arguments. The arguments are rule variables, denoted by the keyword `pvar`, which should be bound to objects of type `Person`.

```
public rule brother(pvar Person p1, pvar Person p2) {
...
}
```

[1] Note that another operation is deleting an object from the working memory, but this does not activate the inferencing process.

In Kiev, rule variables are constructed as shown in the following example. PVar is a parametric type. For creating a variable object, the template has to be instantiated with a type, in this case String.

```
PVar<String> v = new PVar<String>;
```

5.4 Discussion

Table 5.1 gives an overview of all the hybrid systems included in our survey. There is a column for each identified language integration issue. For each system we indicate which approach it takes to addressing each issue. Note that there will always be subtle differences between the approaches, which are generalised in this table.

A first general observation is the similarity between some systems. There are a number of hybrid production systems which have an almost identical approach. They are grouped together in the blue part of the table. Similarly, most logic-based hybrid systems exhibit the same features, indicated by the green part of the table. Then, there are some systems that have a distinct approach. First, Jess is based on another rule-based language than the other hybrid production systems, more specifically CLIPS instead of OPS5. This aside, it has a very similar way of integrating its two languages. Secondly, CommonRules is a forward chainer but based on logic as opposed to production rules. Its representation of data as predicate-based facts rather than objects is the cause of it being the odd one out. Finally, there is Kiev, which is the only symbiotic system. Its main characteristic is a seamless integration between Java and Prolog-based language features.

The two main groups shown in Tab. 5.1 each have a distinct integration philosophy, mainly due to the fact that the first group integrates with a production rule language and the second group with a logic-based language. Typically, a logic-based language is extended with special predicates which map onto message sends. The current hybrid systems based on production rules, however, extend the production rule language itself with new constructs that allow the explicit representation of message sends. Hence, instead of providing an interface for integration with the object-oriented language, the production rule language itself becomes an integration of rule-based and object-oriented language features. As such, they no longer merely match and access data, but are able to activate object-oriented behaviour. Some hybrid systems based on production rules take this even further and express the conclusion of production rules in object-oriented code which is entirely evaluated by the object-oriented language. This, however, means that production languages found in hybrid systems such as OPSJ and JRules tend to be quite different from traditional production rule language such as provided by OPS5. The integration philosophy of hybrid production systems makes obliviousness at the language level impossible to achieve from the start, because object-oriented language features are included in the production rule language. Nevertheless, because hybrid production systems are the "new generation" production systems, we consider them in this dissertation.

All systems introduce the notion of an object in the rule-based language and allow it to be bound to rule variables. This ensures language obliviousness with respect to the values being used by both languages, which are objects. The only exception is when unbound rule variables are required in the object-oriented language, for example to activate unbound queries.

However, with respect to the other language integration issues, we conclude that language obliviousness is not achieved. The software engineer who uses the object-oriented language has to be aware of the rule-based paradigm and the semantics of rule-based language features. The same is true for the knowledge engineer: he or she needs to be familiar with the object-oriented programming paradigm and specific object-oriented language features.

We observe that when some degree of language obliviousness in existing hybrid systems is achieved, it is often at the expense of automatic integration. Indeed, transparent integration is usually enabled by a manual specification of how certain program elements of the different languages map. On the other hand, automation typically results in a less flexible integration. This is because automation is enabled by a one-to-one mapping between program elements of the different languages from which cannot be deviated.

Finally, an important lesson learned from investigating these existing systems is the typical points for integrating a rule-based and an object-oriented program:

- rule-based integration points:
  - **sensors**
    Whether the rule-based language is forward-chaining or backward-chaining, based on production rules or logic, a rule condition can be established by invoking a message.
  - **effectors**
    Forward-chaining rules can revert to sending a message when a conclusion is generated — or, in production rule terminology, when an action is performed.
- object-oriented integration points:
  - **message invocation** Object-oriented programs can activate a rule-based program instead of invoking a message for achieving behaviour.

These integration points are crucial for determining the join points in rule-based programs later on.

Table 5.1: An overview of the approach of each surveyed hybrid system for each language integration issue.

	activating rules in objects	returning inference results	objects as rule values	sending messages in rules	returning message send results	rule values in methods	implemen-tation approach
OPSJ	method API	implicit	wrapped	message	implicit	objects	library
Quick Rules	method API	implicit	wrapped	message	implicit	objects	library
Eclipse	method API	implicit	wrapped	message	implicit	objects	library
Blaze Advisor	method API	implicit	wrapped	message	implicit	objects	library
JRules	method API	implicit	objects	message	implicit	objects	library
NéOpus	method API	implicit	objects	message	implicit	objects	library
Jess	method API	implicit	wrapped	predicate API	explicit	fact objects	library
Common Rules	method API	implicit	deconstructed	mapping	explicit	term objects	library
SOUL	method API	streamed	wrapped	environment	implicit	term objects	library
K-Prolog	method API	streamed	wrapped	predicate API	explicit	objects	symmetric
B-Prolog	method API	streamed	wrapped	predicate API	explicit	term objects	library
Prolog Cafe	method API	streamed	wrapped	predicate API	explicit	term objects	library
Jinni	method API	streamed	wrapped	predicate API	explicit	term objects	library
Quintus Prolog	method API	streamed	wrapped	predicate API	explicit	term objects	symmetric
SICStus Prolog	method API	streamed	wrapped	predicate API	explicit	term objects	symmetric
JavaLog	environment	collection	wrapped	predicate API	explicit	objects	library
Inter Prolog	method API	deterministic	deconstructed	predicate API	explicit	term objects	library
Kiev	transparent	streamed	objects	message	implicit	parametric types	symbiotic

Chapter 6

Model of Hybrid Composition and Hybrid Aspects

This chapter amalgamates the topics we introduced and investigated in the previous chapters. It addresses the integration of object-oriented functionality and rule-based knowledge, each represented in their corresponding language of a hybrid system, in an oblivious way. We introduce a model of our aspect-oriented programming approach, a combination of *hybrid composition* and *hybrid aspects*. Note that we report on the foundations for hybrid composition, *linguistic symbiosis*, in [D'Hondt et al., 2004] and on hybrid aspects in [D'Hondt & Jonckers, 2004].

First, we reflect on our problem domain, and the constraints and variabilities it imposes on the solution domain in section 6.1. Then we present the criteria by which potential solutions can be evaluated in section 6.2. Section 6.3 elaborates on how a composition-based and an aspect-based approach complement each other. Since a combination of these aspect-oriented approaches requires that they use the same *join point model*, this is investigated in section 6.4.

After these preliminary sections, we introduce the core of our model in the next three sections. First, section 6.5 presents our join point model, which considers both object-oriented and rule-based languages. Then, hybrid composition and hybrid aspects are explained for different flavours of rule-based languages in section 6.6 and 6.7, respectively. Finally, we draw conclusions about our approach in section 6.8.

6.1 Problem and Solution Domains

A problem statement typically constrains a solution and introduces variability in a solution. In this dissertation, the problem domain constrains the development context to hybrid systems, thus disregarding an purely object-oriented development context. Additionally, the problem domain considers different rule-based formalisms and chaining strategies, which introduce variability, reflected in the solution domain. Furthermore, we wish our solution to exhibit certain characteristics, such as obliviousness, which guides our choice of approaches: we have shown in chapter 3 that aspect-oriented programming achieves obliviousness.

We attempt to visualise this in figure 6.1, where the top left box encompasses the solution domain and the bottom right one the problem domain. Each domain consists of certain parts: *aspect-oriented approach* and *join point model* in the solution domain; *development context*, *rule-based formalism* and *rule-based strategy* for the problem domain. Each of these parts can be concretised differently and the options are listed next to each part. The options that are not considered are crossed out (in red). When more than one

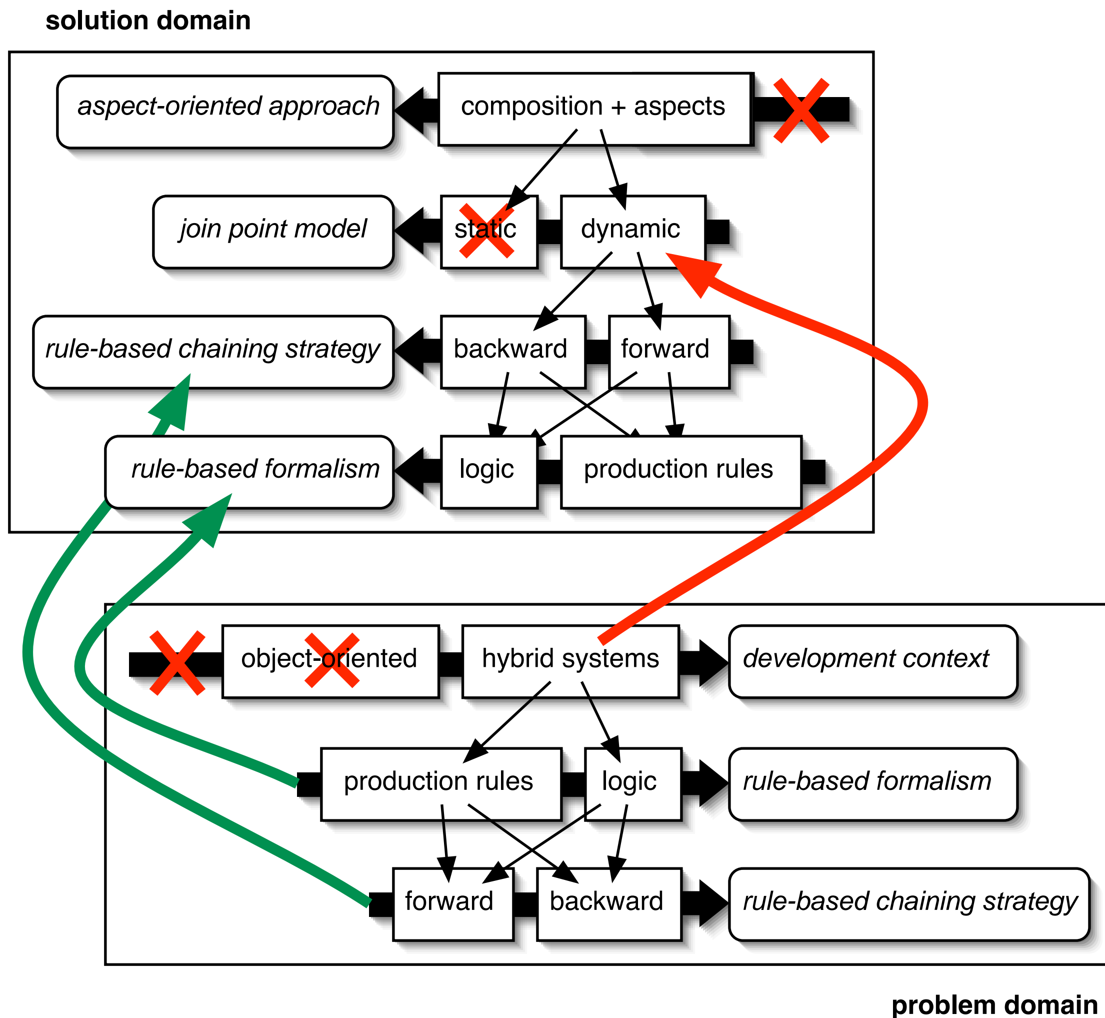


Figure 6.1: Constraints and variabilities imposed by the problem domain on the solution domain.

option is available per part this indicates variability. Rule-based formalism and chaining strategy are variabilities. The variability in the problem domain is reflected in the solution domain, illustrated with the fat (green) arrows from the rule-based formalisms and chaining strategies in the problem domain to the solution domain. The narrow black arrows indicate the valid paths between the options of different parts. For example, an aspect-oriented programming approach has two options for its join point model: static or dynamic. The problem domain constrains these particular options, because a hybrid development context dictates the use of a dynamic join point model, as explained later. In the problem domain, each path of arrows followed from top to bottom forms a concrete problem in the family of problems. The same holds for the solution domain. In this chapter, we aim to discuss the family of solutions which constitutes our approach as independently as possible from specific languages and implementations.

6.2 Criteria

In this section, we set the criteria by which potential solutions can be evaluated. We distil a set of criteria from the two sets of integration issues we identified earlier in this dissertation:

1. There are the **program integration issues**, concerned with *obliviousness at the program level* of object-oriented functionality and rule-based knowledge (chapter 2). We have shown that existing aspect-oriented programming approaches are able to address these issues when the rule-based knowledge is represented as rule objects (chapter 3).

2. There are the **language integration issues**, concerned with *obliviousness at the language level* of object-oriented and rule-based languages in hybrid systems (chapter 5). Hybrid systems provide two languages from different paradigms instead of the single object-oriented programming language often used in modern-day software development. This poses additional challenges.

Addressing some of these issues requires the conception and development of fundamentally new (aspect-oriented) techniques. These issues are adopted as criteria. When evaluating our approach in chapter 8, we show how our approach addresses these criteria. One of the original program integration issues, combining rules, becomes obsolete in a hybrid systems context (section 6.2.5).

6.2.1 Symmetric Obliviousness at the Program Level

When a program is oblivious to its integration with other programs, the dependencies between this program and the others are minimised. Originally we identified the need for obliviousness in order to avoid dependency of the object-oriented functionality on the rule-based knowledge when the former activates the latter at certain events. However, the introduction of hybrid systems as a development context makes it clear that obliviousness can go either way: rules also have to be oblivious to any object-oriented functionality they are integrated with.

A particular challenge in achieving obliviousness at the program level is when data is required at a certain point by either the rule-based or the object-oriented program, but unavailable at this point. In order to preserve obliviousness, a mechanism is required that allows capturing and passing the unavailable data without having to change the object-oriented functionality.

> *An approach to integrating rule-based knowledge and object-oriented functionality in a hybrid system has to pursue symmetric obliviousness at the program level, such that both rule-based knowledge and object-oriented functionality are oblivious to the integration.*

6.2.2 Symmetric Obliviousness at the Language Level

As we have seen in existing hybrid systems, both object-oriented and rule-based language are typically extended with features from the other language in order to achieve language integration. These paradigm leaks make the integration between rule-based and object-oriented programs even more explicit. Furthermore, there is no compatibility between the original and the extended language: programs implemented in the latter cannot run in the former. The same is possibly true for compatibility in the other direction. Another problem

is that programmers who are adept at using one of the original languages, suddenly need to consider the features of the other programming paradigm the language is extended with. More often than not, these new features interact with the features they are accustomed to, which makes the latter behave in unexpected ways.

> *An approach to integrating rule-based knowledge and object-oriented functionality in a hybrid system must achieve symmetric language obliviousness, which means that both the object-oriented language and the rule-based language in the hybrid system must be oblivious to the integration.*

Note that this criterion requires all six language integration issues to be considered.

6.2.3 Default and Customised Integration

We observe that when some degree of language obliviousness in existing hybrid systems is achieved, it is often at the expense of automatic integration. Indeed, some existing hybrid systems achieve true transparent integration by means of a manual specification of how certain elements of programs implemented in the different languages map. However, this manual approach can be cumbersome if for each integration point a separate mapping has to be defined. Conversely, the programs themselves can contain elements that are recognised by the evaluator as integration points with programs in the other language. We have seen that typical approaches are the introduction of new language constructs or the use of special keywords for representing these integration points. This approach results in an automatic integration because the language evaluator is able to detect the integration points and infer what they are integrated with in the other language. However, this automatic integration is less flexible than the manual one. This is because automation is enabled by a universal, one-to-one mapping between program elements of the different languages from which cannot be deviated.

> *An approach to integrating rule-based knowledge and object-oriented functionality in a hybrid system has to have the best of both worlds: support for an automatic default integration and support for overriding this integration in a highly customised way.*

Naturally, such a combined integration approach must support the aforementioned symmetric obliviousness at the language and program level.

6.2.4 Encapsulated Integration Code

In the simplest case, an integration of rule-based knowledge and object-oriented functionality consists of the activation of the one at certain points in the other. Yet, a discrepancy is possible between the behaviour that is expected by the activating program and the behaviour that is provided by the activated program. This is due to the integration being unanticipated or variability being present between different activators of the same behaviour. In any case, these incompatibilities are typically resolved by writing extra code that manipulates the result of the activated behaviour and/or activates additional behaviour. We refer to this extra code as *integration code* because it does not belong to either one of the programs but is specific to a particular integration.

In current hybrid systems, any integration code is expressed as either extra object-oriented or rule-based code. As a result, either object-oriented functionality or rule-based

knowledge is cluttered with integration code that is only relevant to one particular integration. Moreover, the integration code, expressed in one of the languages, contains a high concentration of features from the other language, which the first is extended with. We say the integration code is hybrid.

> *An approach to integrating rule-based knowledge and object-oriented functionality in a hybrid system has to provide support for encapsulating the possibly hybrid integration code and separate it from both object-oriented functionality and rule-based knowledge.*

6.2.5 Obsolete Issue: Combining Rules

The responsibility of one of the original program integration issues shifts from being a concern of the integration to the rule-based language in a hybrid system: *combining rules*. Addressing this issue in an object-oriented or even aspect-oriented programming environment is cumbersome, as shown in chapter 3. One of the main reasons for dealing with the combination of rules in rule-based programs themselves, is that rule-based languages offer support for expressing rule combinations. Rule-based languages deal with this in different ways, such as supporting the definition of meta rules which influence the way rules are combined or allowing the association of priorities with the rules themselves. Therefore, this issue does not become obsolete in general, it merely does no longer have to be considered when integrating rule-based and object-oriented programs. It is still very much relevant when representing rule-based knowledge in a rule-based language.

6.3 Composition and Aspects are Complementary

We have shown in chapter 3 that the initial set of integration issues is addressed by aspect-oriented programming approaches. Therefore, our approach is also based on aspect-oriented programming and, particularly, combines a form of aspect-oriented programming based on *composition*, such as HyperJ, and one based on *pointcut/advice*, such as AspectJ and JAsCo. The terms composition and pointcut/advice are coined in [Masuhara & Kiczales, 2003]. Note that from now on, we use the term *aspect*-based approaches whenever we want to refer to pointcut/advice-based approaches. We call the integration of programs in composition-based approaches *composition* and *weaving* in aspect-based approaches. The term “aspect-oriented (programming) approach” still encompasses all kinds of approaches, whether they are composition-based, aspect-based or any other.

This section shows in a high-level way that composition-based and aspect-based approaches complement each other. The ensuing discussion does not depend on the development context — object-oriented or hybrid — nor on the join point model — static or dynamic. In order to achieve this, we use the phrases “means for identifying join points” and “means for effecting semantics”, introduced in the modelling framework of [Masuhara & Kiczales, 2003].

Approaches based on composition and aspects complement each other. Composition enables the definition of a universal strategy for composing two programs automatically, based on a mapping between program or execution elements. Aspects, on the other hand, allow a very selective and customised integration of programs, by specifying a set of heterogeneous join points and defining advice that is to be weaved at these join points. In order to compare the two approaches more easily on a conceptual level, we give a graphical representation in figure 6.2 of how both approaches can be used to integrate two programs. This figure is used in the ensuing discussion.

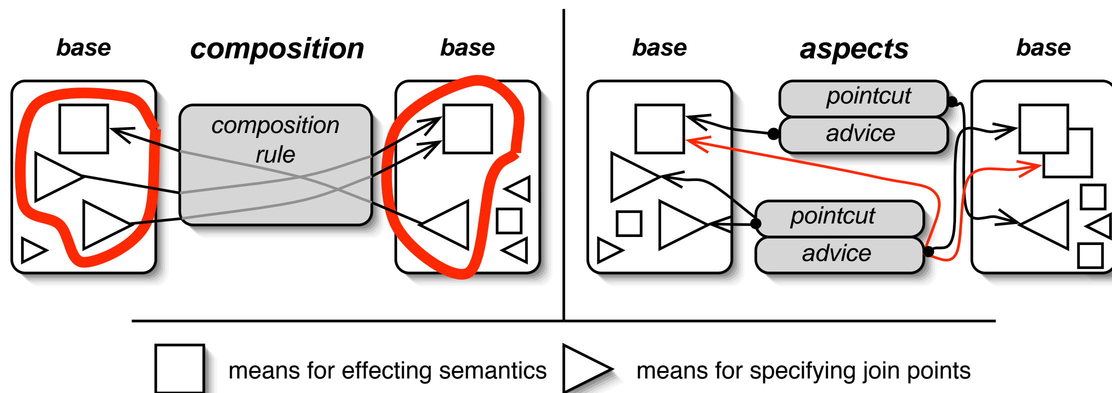


Figure 6.2: Comparison between composition-based and aspect-based approaches.

6.3.1 Elements of the Base Input Programs

In figure 6.2 two integration approaches of two base input programs are shown, the one on the left-hand side using composition and the one on the right-hand side using aspects. Each base program is implemented in a base language, which, in current aspect-oriented programming approaches, is usually an object-oriented programming language. In the context of hybrid systems the base language can either be an object-oriented language or a rule-based language. The base languages consist of elements that correspond to join points and of elements that effect semantics at join points. In figure 6.2, the former are depicted as triangles and the latter as squares. Effecting semantics can be achieved by activating behaviour or providing the declaration of a program element, depending on how the aspect-oriented programming approach is conceived. As we will see later on, this choice depends on the kind of join point model used, static or dynamic. Examples of activating behaviour are message sends in object-oriented languages, and queries or asserts in rule-based languages. Examples of program element declarations are method and rule definitions in object-oriented and rule-based languages, respectively.

6.3.2 Characteristics of Composition

Figure 6.2 shows that composition consists of a specified composition rule which denotes the general composition strategy to be employed for this particular integration. Although only one composition rule needs to be specified, this results in multiple mappings between automatically identified join points and means for effecting semantics in the two base programs being integrated. Nevertheless, bear in mind that composition in a hybrid system should be able to compose automatically identified join points of a program in one base language onto program elements that effect semantics implemented in the other base language.

Furthermore, a composition-based approach has a way of constraining the scope of the composition to some parts of the input programs. This is illustrated in figure 6.2 with the irregular (red) contours that enclose certain elements in each base program but leave others out. For example, it is possible that rule-based knowledge should only be composed with the core functionality of a software application and not with certain parts that deal with technical issues such as persistence, transaction management and so on. Alternatively, it is possible that certain rules are not relevant to a part of the object-oriented functionality. Such restrictions boost efficiency since searching for matches inevitably introduces a certain

overhead, either before or at run-time, even if no actual composition is performed. Moreover, restricting the composition avoids possible accidental compositions in parts of the program that are not meant to be composed.

6.3.3 Characteristics of Aspects

Figure 6.2 shows that aspect-based integration of two base programs requires the specification of an aspect for each integration. The pointcut of each aspect possibly specifies several heterogeneous join points at which the advice will be weaved, but this typically does not constitute a general integration strategy. Intuitively, a weaver takes one program implemented in the (only) base language and an aspect program as input. The advice consists, among others, of statements that activate behaviour of the base program. In order to obtain an architecture that is comparable to the one of composition-based approaches, think of the weaver taking two base programs as input together with one or more aspects for integrating them. The integration is enabled when an aspect's pointcut specifies join points in one base program and the corresponding advice activates behaviour in the other base program. This is illustrated in figure 6.2. Of course, this set up is also very suitable in the context of hybrid systems, where two base input programs are integrated, a rule-based and an object-oriented program.

A major advantage of aspects, which makes them complementary to composition, is that advice is a program containing statements for invoking additional behaviour for customising the integration. This means that aspects are able to encapsulate any integration code which might be needed for adapting the base programs to one another. This is illustrated in figure 6.2 by the additional (red) arrows which start from the lower aspect's advice.

6.3.4 Differences between Composition and Aspects

It is important to note that sophisticated composition-based aspect-oriented approaches, such as HyperJ, are also able to specify a selective set of join points in addition to the general composition strategies. Moreover, HyperJ has additional features, referred to as *bracketing*, for composing program elements in a specific order (one before or after the other). However, HyperJ does not have the equivalent of a dedicated aspect language which encapsulates the specification of bracketing and additional means for effecting semantics. HyperJ is able to express this, but needs to combine an extra input program containing the additional means for effecting semantics.

Likewise, sophisticated aspect-based approaches, such as AspectJ and JAsCo, are able to specify extremely general pointcuts which capture each possible join point of an input program. However, the advice associated with such a pointcut is not able to express that each of the captured join points should be mapped to a corresponding means for effecting semantics. Indeed, the advice expresses a common behaviour that should be executed at each of the join points, albeit parametrised with values from the context of each join point.

As such, we only consider the basic, distinguishing features of each kind of aspect-oriented programming approach, which are indeed complementary, and combine those.

6.4 Dynamic vs Static Join Point Models

Aspect-oriented approaches use either a static or a dynamic *join point model* (section 6.4.1). We argue that in the context of hybrid systems, a dynamic join point model is more appropriate (section 6.4.2).

6.4.1 Static vs Dynamic Join Point Models

Approaches that have a static join point model manipulate the input programs themselves, looking for join points that correspond directly to statements in the programs and integrating declarations of program elements with them. Approaches with a dynamic join point model, on the other hand, manipulate the *execution* of the input programs, looking for join points that are points in the execution of the programs. These join points are usually reflected — or have a *shadow* — in the program definition. Integration consists of activating some behaviour at the identified join points instead of providing program declarations.

6.4.2 Suitable Model for Hybrid Systems

For our approach we select a dynamic join point model and the choice is mainly influenced by the development context being hybrid systems. Hybrid systems consist of two different languages, in our case an object-oriented and a rule-based language. Note that at this point we do not need to know the particular rule-based formalism or chaining strategy used. Existing hybrid systems typically integrate programs in terms of the execution of the programs rather than merge the program definitions themselves. Indeed, most hybrid systems provide some way of passing control from one language evaluator to the other. Note that in most cases, the actual integration also happens at run-time. Only a few systems provide a compiled program as the result of integration.

When both composition and aspects employ a dynamic join point model, they both consider dynamic join points and the same strategy for effecting semantics at the join points, i.e. executing behaviour. When effectively combining composition and aspects, the *kinds* of dynamic join points existing in the model should be unified in order to obtain a well-defined, common set of join points to be employed by both approaches. When this is the case, certain strategies can be envisaged that combine or override composition and aspects at particular join points. We discuss the unified dynamic join point model for rule-based and object-oriented languages in the next section, together with how behaviour implemented in these languages is activated.

6.5 Base Languages

Before we present hybrid composition and hybrid aspects, let us introduce their base languages. An important difference with traditional aspect-oriented approaches is that two base languages have to be considered instead of one — an object-oriented and a rule-based language. The way all aspect-oriented approaches with dynamic join point models typically operate is to interrupt the execution of a base program at a certain join point, and activate some behaviour instead. Therefore, we need to determine join points and behaviour activation in both object-oriented and rule-based languages. In order to distinguish between distinct join points and behaviour activation, these elements need to be qualified.

In the following, we first discuss behaviour activation, join points and qualifiers for object-oriented languages (section 6.5.1). After that, we do the same for rule-based languages (section 6.5.2).

6.5.1 Object-Oriented Base Languages

Since this work is a first attempt at integrating object-oriented functionality with rule-based knowledge in hybrid systems using aspect-oriented programming, we only consider a pure object-oriented programming model. In such a simple, yet complete model, everything is an

object and all behaviour is activated with message passing. Moreover, as mentioned earlier in this dissertation (section 5.1), we only take class-based inheritance into account.

Object-Oriented Behaviour Activation

In a pure object-oriented programming language, behaviour activation is achieved by **sending messages**. Even instantiating new objects from a certain class is achieved by sending messages to the object representing the class.

Object-Oriented Join Points

There are several points in the execution of object-oriented programs that are appropriate join points and most centre around methods. When considering dynamic join point models used by existing aspect-oriented programming approaches, we note that recurring join points are method call, method reception and method execution. The main difference between these kinds of join points is that the context of a method call is the caller object whereas the context of a method reception or execution is the callee object. The distinction between the latter two is that method reception is before the method lookup and method execution is when the looked up method is actually executed. To keep our dynamic join point model simple, we consider **method execution** as the only kind of object-oriented join point.

However, hybrid composition for production rule languages is an exception. Because production rules are data-oriented, they should be automatically activated when object attributes are accessed or mutated. Therefore, for this particular case, we consider attribute access and mutation join points as well.

Object-Oriented Qualifiers

Hybrid composition and hybrid aspects each integrate join points with behaviour activation in their own particular fashion. In both cases, however, identification of these elements is required. In our model, object-oriented join points are method executions and object-oriented behaviour is activated with message sends. Both of these elements are qualified in a similar yet distinct way. Let us consider both kinds of qualifiers separately.

Object-Oriented Join Point Qualifiers can consist of any property that can be used to distinguish one specific method execution from another, such as:

1. the class that defines or inherits the method being executed
2. the method name
3. number of parameters
4. type of parameters and return value

The first two properties are usually sufficient to distinguish between executions of different concrete method definitions if the language does not support method overloading. In keeping with the spirit of our approach, we select basic object-oriented join point qualifiers: (1) the name of the class that defines or inherits the method and (2) the method name.

Object-Oriented Behaviour Activation Qualifiers consist of only the message selector since this is the only identification of a message send statement. Indeed, if one considers a typical object-oriented message send statement, $o.selector(p_1, ..., p_k)$, the receiver o and parameters $p_1, ..., p_k$ are typically variables which are not qualified at all. The only distinguishing element is the message selector.

6.5.2 Rule-Based Base Languages

There are many differences between rule-based languages, aside from the differences in rule-based formalism and chaining strategy. Nevertheless, similar to the object-oriented programming model, we only take basic features of rule-based languages into account. The representative of the logic-based languages considered in this dissertation is Prolog. Originally, the production rule language OPS5 was the ambassador for languages of this formalism. In modern-day hybrid production systems, however, the traditional production rule languages are adapted substantially. Before we present rule-based behaviour activation, join points and qualifiers, we discuss traditional versus “new generation” production rule languages.

Important to note is that all rule-based languages regarded in this model take objects as only rule value. In the same spirit as the object-oriented languages considered in our model, even literals such as numbers and strings are objects in the rule-based languages. Therefore, a rule variable is either unbound, or bound to one or more objects. Such rule-based languages have unification or pattern matching algorithms that are extended to take objects into account. Since rule-based languages in most existing hybrid systems support this, it is a reasonable prerequisite for our approach.

Traditional vs New Generation Production Rule Languages

There is a substantial difference between the traditional production rule languages, such as OPS5, and the new generation of production rule languages found in hybrid production systems, such as OPSJ and JRules. Current hybrid systems based on production rules typically extend the production rule language itself with new constructs that allow explicit representation of message sends. Hence, the production rule language itself becomes an integration of rule-based and object-oriented language features. As such, they no longer merely match and access data, but are able to activate object-oriented behaviour explicitly. Let us review both kinds of production rule languages separately.

Traditional Production Rule Languages

In traditional production systems, the rules consist of patterns to which data in the working memory are matched, using data type and attributes. In the rule’s conclusion, or action, operations are performed on the matched data which modify attributes, create or delete data. For example, consider the OPS5 production rule below[1].

```
(p loyalCustomer
  (Customer ^totalPurchases <tot> ^age <a>)
  (<a> > 18)
  (<tot> > 100)
do
  (modify 1 ^status loyal))
```

New Generation Production Rule Languages

A typical new generation production rule, such as the one found below, uses a syntax extended with object-oriented statements. We refer to these statements as *condition message sends* and *conclusion message sends*. The condition consists of two kinds of statements: statements for matching working memory elements to rule variables (between the delimiters `|` and `|`) and condition message sends. In this particular

[1]Note that the syntax of OPS5 is explained in the chapter on rule-based systems (chapter 4)

(fictituous) syntax, a message send statement consists of a receiver — typically a rule variable — a dot and the message. The message consists of an identifier and a list of arguments between brackets. In a condition, the result of a message send can be matched to a rule variable, in this case with the operator `=`. If this operator is not used, the message should evaluate to true or false, as exemplified in the first condition message send. Note that the keyword `do` for separating conclusions and conditions is replaced by `->`.

```
(p loyalCustomer
  | <c> isa Customer |
  <c>.isAdult(),
  <tot> = <c>.getTotalPurchases(),
  <tot> > 100
->
  <c>.setStatus('loyal'))
```

However, the backward-chaining variety of these production rules, which infers attributes of objects, requires a slightly distinct conclusion. When a production rule query is activated for the value of a certain object attribute, the engine needs to be able to look up the rules that are able to infer the attribute. With conclusion message sends, it is not possible to detect if a particular attribute will be inferred. Therefore, the backward-chaining production rules we consider are similar to the one given below. In the conclusion, `status` is the name of the attribute, `<c>` and `'loyal'` are values, and `of` and `is` are keywords.

```
(p loyalCustomerBC
  | <c> isa Customer |
  <c>.isAdult(),
  <tot> = <c>.getTotalPurchases(),
  <tot> > 100
<-
  status of <c> is 'loyal')
```

We observe that the arguments of a condition or conclusion message send are all bound.

Although it is possible to consider the traditional production rule languages in our work, we only take the more recent hybrid production rule languages into account. A first reason is that the latter embody the current state of the art and can be regarded as the new generation of production rule languages. Furthermore, they take integration with object-oriented languages explicitly into account, which provides an interesting contrast with the usual approach of logic-based hybrid systems. However, hybrid production rule languages obviously do not allow us to achieve obliviousness at the language level of the production rule language, because it incorporates object-oriented language features from the start.

Rule-Based Behaviour Activation

Backward-Chaining Behaviour Activation is triggered by performing a **query**. Typically, backward-chaining languages understand an assert of a new fact as well, but this operation does not trigger inferencing.

Forward-Chaining Behaviour Activation is typically triggered with an **assert**. However, in some forward-chaining languages, an assert simply adds a fact without activating the inference process. A separate operation is provided for triggering forward-chaining inferencing. We only consider the first kind of asserts for forward chainers. Analogous to backward chainers, forward-chaining languages are also able to process queries. However, different languages interpret such a query in different ways. First of all, a query can simply retrieve inferred facts. Such a query does not cause the forward chainer to start inferring, it merely returns already inferred facts. Secondly, a query with a specific goal can trigger forward chaining, which stops if the goal is inferred. Once more, we only consider forward chainers that are activated by asserts. Optionally, they provide a simple query for retrieving inferred facts.

Rule-Based Join Points

Since the join point model is dynamic, the join points denote points in the execution of rule-based programs. There is a distinction between forward chaining and backward chaining languages, although there is also a partial overlap.

In establishing the join points in a rule-based language, we consider the typical integration points in rule-based programs of existing hybrid systems. As we concluded in chapter 5, these are *sensors* and *effectors*. The former is the activation of object-oriented behaviour when a condition is being inferred in both backward and forward-chaining rules. Note that in backward chainers, inferring a query is also considered to be such a condition inference join point. The latter is the activation of object-oriented behaviour when a conclusion is being generated in forward-chaining rules. This results in the following kinds of rule-based join points:

Backward-Chaining Join Points are **condition inferences**.

Forward-Chaining Join Points are either **condition inferences** or **conclusion generations**.

Rule-Based Qualifiers

Rule-based qualifiers do not depend on the chaining strategy, but on the rule-based formalism of the language. Therefore, we distinguish between logic-based and production rule qualifiers.

Logic-Based Qualifiers for both join points and behaviour activation are predicate name and, optionally, arity. Most logic-based languages regard predicates with the same name yet different arities as distinct predicates. However, some logic-based languages support predicates with variable arities. We take only logic-based languages into account that consider predicate arity to be a distinguishing property. Note that similar to object-oriented qualifiers, logic-based join points can be qualified by the type of the arguments. This means that, for example, specific condition inferences can be distinguished from one another based on the types of their arguments, and not only based on the predicate. However, we only consider basic logic-based qualifiers in our model: predicate name and arity.

Production Rule Qualifiers are different for join points and behaviour activation.

Production Rule Join Point Qualifiers are similar to object-oriented join point qualifiers, since condition and conclusion message sends are used in new

generation production rules, which result in methods being executed. We repeat the parts of such qualifiers:

1. the class that defines or inherits the method being executed
2. the method name

Production Rule Behaviour Activation Qualifiers are attribute names. Since an attribute of an object can be asserted or queried, the name of the attribute has to specified. Optionally, the class name of the object can also be part of the qualifier, although we do not consider that in our model.

6.6 Hybrid Composition

This section introduces our model of hybrid composition which integrates object-oriented functionality and rule-based knowledge. Hybrid composition employs join points and behaviour activation in object-oriented and rule-based languages, which were introduced in the previous section.

This section first presents the general approach considered by hybrid composition in section 6.6.1. Since hybrid composition needs a linguistic symbiosis between object-oriented and rule-based languages, this is explained in section 6.6.2 for logic-based languages and in section 6.6.3 for production rule languages.

6.6.1 General Approach

When composing two object-oriented programs, an object-oriented join point — such as a method execution — corresponds to the only means for activating object-oriented behaviour — sending a message. This unambiguous correspondence partly automates the composition. The other required part for automating the composition is a one-to-one match between a specific join point and a specific behaviour activation. This match uses join point and behaviour activation qualifiers. Consider again composition of two object-oriented programs: a method execution matches unambiguously with a message send if the method name equals the message selector.

Hybrid composition composes programs implemented in rule-based and object-oriented languages. Therefore, rule-based behaviour is activated at an object-oriented join point and object-oriented behaviour is activated at a rule-based join point. In order to attain automatic hybrid composition, we also need to determine an unambiguous correspondence and a one-to-one match between object-oriented join points and rule-based behaviour activation and vice versa.

Correspondence between Join Points and Behaviour Activation

Rule-based join points, whether they are condition inferences or conclusion generations, are always composed with a message send, since this is the only kind of object-oriented behaviour activation. This is shown in table 6.1.

The correspondence in the other direction is not so clear-cut. Rule-based languages, whether backward-chaining or forward-chaining, understand both queries and asserts. Thus, we need to select with which one of these two operations object-oriented join points correspond. In section 6.5.2, we state that although rule-based languages understand both queries and asserts, we only consider rule-based languages where just one of these operations effectively activates inferencing: backward-chaining is activated with a query whereas

Table 6.1: Unambiguous correspondence used by hybrid composition between rule-based join points and object-oriented behaviour activation.

rule-based join point	→	**object-oriented behaviour activation**
backward-chaining		
condition inference		message send
rule-based join point	→	**object-oriented behaviour activation**
forward-chaining		
condition inference		message send
conclusion generation		

forward-chaining is activated with an assert. Therefore, an object-oriented join point corresponds unambiguously to a query when integrating with a backward-chaining language whereas an object-oriented join point corresponds to an assert when integrating with a forward-chaining language.

But we need to make a distinction between the object-oriented join points at which logic-based behaviour is activated and at which production rules are activated. Usually, we consider method execution to be the only object-oriented join point in our model. This works for hybrid composition between object-oriented and logic-based programs, as shown in table 6.2. However, because production rules are data-oriented, they should be automatically activated when object attributes are accessed or mutated. Therefore, for this particular case, we consider attribute access and mutation join points as well. This is illustrated by table 6.3.

Table 6.2: Unambiguous correspondence used by hybrid composition between object-oriented join points and logic-based behaviour activation.

object-oriented join point	→	**logic-based behaviour activation**
		backward-chaining
method execution		query
object-oriented join point	→	**logic-based behaviour activation**
		forward-chaining
method execution		assert

Table 6.3: Unambiguous correspondence used by hybrid composition between object-oriented join points and production rule behaviour activation.

object-oriented join point	→	**production rule behaviour activation**
		backward-chaining
attribute access		query
object-oriented join point	→	**production rule behaviour activation**
		forward-chaining
object instantiation		assert
attribute mutation		

Matching between Join Points and Behaviour Activation

Using the above tables, hybrid composition is able to determine which *kind* of behaviour should be activated at a certain join point — a message send, a query or an assert — but not yet which concrete one. To achieve this, a mechanism has to be developed that is able to determine the exact behaviour activation that matches a concrete join point. As we mentioned earlier, qualifiers enable this required one-to-one match between corresponding join points and behaviour activation. Since this requires considering qualifiers of two languages of different paradigms, this is not so straightforward: a *linguistic symbiosis* has to be established between the two languages. For logic-based and object-oriented languages this is described in the next section (section 6.6.2); we explain the same for production rule languages and object-oriented languages in section 6.6.3.

Alternative Composition Strategies

Once hybrid composition determines the exact behaviour to be activated at a certain join point, different composition strategies can applied:

- the new behaviour determined by hybrid composition can be executed **before** or **after** the original behaviour interrupted at the join point

- the new behaviour determined by hybrid composition can be executed **instead of** the original behaviour interrupted at the join point

- a special case of the previous strategy is that hybrid composition only determines and executes new behaviour if there is no original behaviour available at the join point. This situation arises when the join point represents a method execution but the method is not defined, or when the join point represents a condition inference but no candidate rules or facts are available for inferring the condition. In some rule-based languages, a conclusion can always be generated because this does not depend on program elements being defined beforehand or not. Hence, at a conclusion generation join point both the conclusion can be generated and the matching message can be sent. In this case, the strategy has to decide which behaviour activation is overridden by the other.

In chapter 7 we implement this last composition strategy. However, in the concluding chapter (chapter 9) we argue in the section on future work that alternative composition strategies can be implemented instead without requiring fundamental changes to the current setup (section 9.2).

Constraining Hybrid Composition

Hybrid composition allows the developer to specify the scope of the composition. Enumerating packages containing classes and rule layers containing rules, allows constraining the input programs considered by hybrid composition.

Diagram and Naming Conventions

In the remainder, we resort to illustrating the linguistic symbiosis schematically. Figure 6.3 shows the shapes and naming conventions we employ in these diagrams.

We review the numbered boxes in the figure:

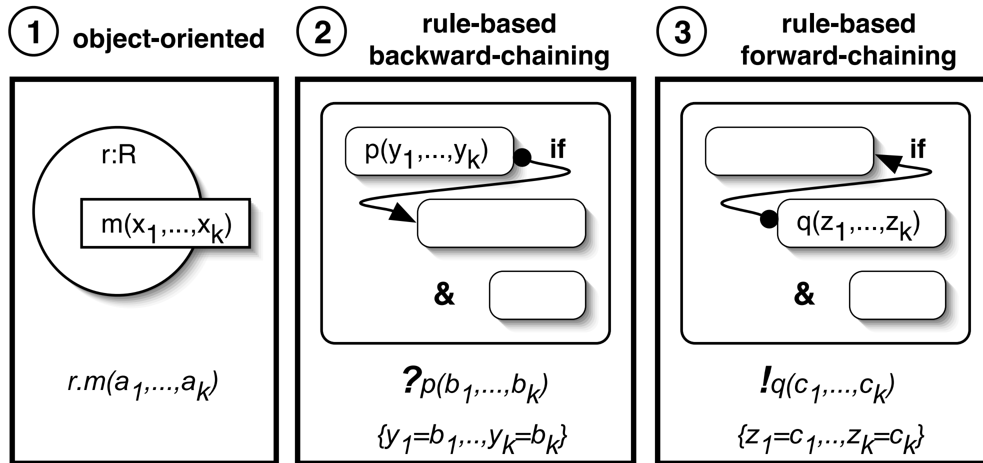


Figure 6.3: Diagram and naming conventions.

1. An object r of class R is depicted by a circle. A method named m with formal parameters $x_1, ..., x_k$, defined in class R or one of its superclasses, is denoted by a box overlapping the circle that denotes an instance of R. The message send $r.m(a_1, ..., a_k)$ has a message selector m, receiver r and actual parameters $a_1, ..., a_k$.

2. The outer box with rounded corners denotes a backward-chaining rule. It consists of one or more conclusions — although we always show only one — followed by the keyword **if** and one or more conditions. Conclusions and conditions are depicted by smaller boxes with rounded corners. A predicate may or may not be specified in these boxes. Conditions are separated by the character **&**. Since it is a backward-chaining rule, there is an arrow pointing to the conditions from the conclusion. A predicate consists of a predicate name p and arguments $y_1, ...y_k$. A query is preceded by a question mark **?**, such as $?p(b_1, ..., b_k)$, which has predicate name p and arguments $b_1, ..., b_k$. When rule variables $y_1, ..., y_k$ are bound to the values $b_1, ..., b_k$, this is represented with the set of bindings $\{y_1 = b_1, .., y_k = b_k\}$.

3. A forward-chaining rule is depicted similarly to a backward-chaining rule. The only difference is that the arrow points from the conditions to the conclusion. An assert is preceded by an exclamation mark **!**.

The naming conventions are: r and s are instances of classes R and S, respectively; method and predicate names are m, n, p and q; formal parameters of methods are w, x, y and z; actual parameters of methods are a, b and c; indices of the parameters and arguments are k and l. Note that all predicate arguments, whether in a rule, a query or an assert, can be rule variables or concrete rule values. Nevertheless, we use w, x, y and z in rule definitions whereas a, b and c tend to be actual values used in queries or asserts.

6.6.2 Linguistic Symbiosis with Logic-Based Languages

Linguistic symbiosis consists of two parts: activating rules from objects and sending messages from rules, which includes related issues such as returning inference and message send results, and rule values in objects. We discuss these two steps for both backward-chaining and forward-chaining logic-based languages below.

This chapter attempts to remain language-independent, but we have implemented the linguistic symbiosis for Smalltalk and a Prolog-like language (backward-chaining logic language) and for Smalltalk and a forward-chaining logic language. In the next chapter, we provide examples of the former linguistic symbiosis in section 7.2.1 and of the latter in section 7.2.2. That way, the reader can refer to the concrete examples in these sections when reading the general and language-independent setup of the linguistic symbiosis in the following.

Activating Backward-Chaining Logic Rules from Objects

Figure 6.4 illustrates how an actual method execution is matched with a concrete query. On the left-hand side, part of an object-oriented program is shown whereas the right-hand side depicts some logic rules.

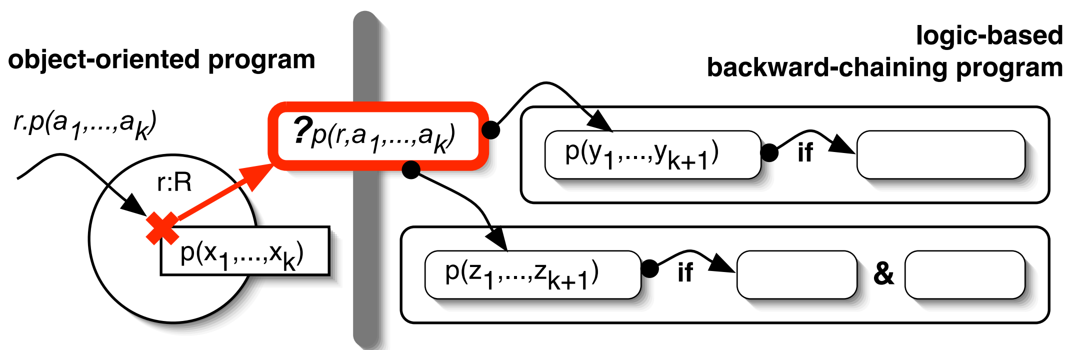


Figure 6.4: Activating backward-chaining logic rules from objects.

At a certain point, a message with selector p and actual parameters $a_1, ..., a_k$ is sent to object r. When the corresponding method named p with formal parameters $x_1, ..., x_k$ is going to be executed, hybrid composition is able to intervene, denoted by the red cross. Depending on the composition strategy, a matching query is activated instead, before or after the original method p. Remember that this method might be altogether non-existent, which simply results in the query being activated. The query's predicate name is p, the name of the original method. Its first argument is the receiver r and the rest of its arguments are $a_1, ..., a_k$, the actual parameters of the method. The query results in possibly multiple rules being triggered which conclude the predicate p with arity $k+1$. In figure 6.4, two candidate rules are triggered. In the case of the lower rule, for example, its arguments $z_1, ..., z_{k+1}$ are unified with the query arguments $r, a_1, ..., a_k$.

Since hybrid composition supports the activation of queries in objects, inference results are expected to be returned, as is normal in the object-oriented paradigm. In the simplest case, a query returns an indication of success or failure which is interpreted as the corresponding boolean value by the object-oriented language. However, more often than not, the object-oriented program expects a value other than a boolean from a behaviour activation.

Existing logic-based hybrid systems resolve this issue by explicitly activating a query from an object-oriented program with an extra argument, which is an unbound variable. The bindings inferred for this variable are retrieved manually from the variable in the object-oriented program. However, anticipating an extra parameter which represents the return value is not compatible with the object-oriented programming paradigm. In order

to deal with this, a mechanism is needed that automatically adds an extra argument to the query and, after inferencing, returns its bindings as the expected return value to the object-oriented program. This is illustrated in figure 6.5: when the method $p(x_1, ..., x_k)$ is going to be executed with receiver r and actual parameters $a_1, ..., a_k$, a query with predicate p and arguments $r, a_1, ..., a_k, w$ can be activated. The last argument, w is an unbound rule variable which is bound if the query is successfully inferred. At this point, hybrid composition takes the bindings for this variable and returns them to the interrupted execution context of the object-oriented program.

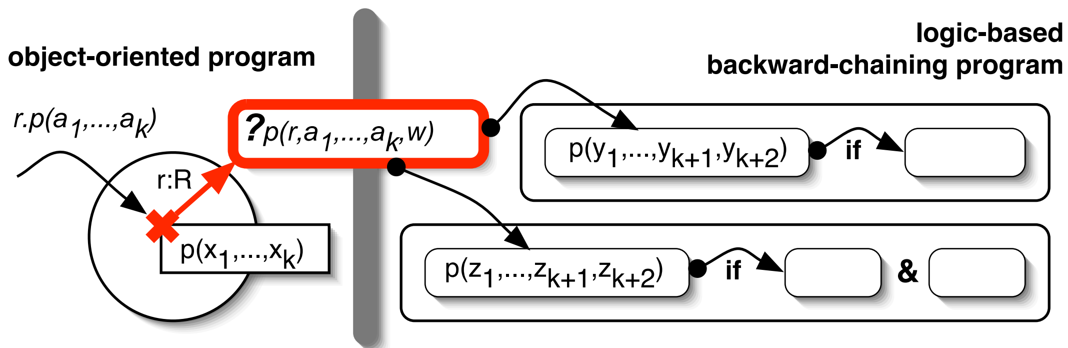


Figure 6.5: Returning inference results from backward-chaining logic rules.

There are several possible strategies for determining when the query $p(r, a_1, ..., a_k, w)$ has to be activated instead of the query $p(r, a_1, ..., a_k)$ previously presented in figure 6.4. In a statically typed object-oriented language, the type of the return value of the method $p(x_1, ..., x_k)$ indicates whether a boolean or non-boolean object is expected. This determines if the matching query $p(r, a_1, ..., a_k)$ has to be activated or the query with extra argument for the return value $p(r, a_1, ..., a_k, w)$.

In dynamically typed languages, a more dynamic approach is required. A possible strategy is first to look for rules which conclude the predicate with name p and arity $k+1$. If such rules exist, the query $p(r, a_1, ..., a_k)$ is activated. If no such rule exists, but at least one rule that concludes the predicate with name p and arity $k + 2$ does, the query $p(r, a_1, ..., a_k, w)$ is activated. Note that when both rules are defined, this strategy might result in unwanted behaviour since the rule which concludes the predicate with the highest arity will never be triggered. Implementing the rule-based and object-oriented programs in such a way as to avoid this, results in programs that are not oblivious to the integration they are part of. As we will show later on in this dissertation, it is more suitable to employ hybrid composition for basic integration between rule-based and object-oriented programs and let hybrid aspects deal with the exchange of values, such as returning inference results.

Despite the above approaches for returning the bindings of an implicit unbound rule variable to the object-oriented program, situations might arise that require the activation of a query with explicit, unbound variables from an object-oriented program. In particular, these situations occur when more than one unbound variable is needed or when the only argument of the query should be unbound. In order to support this, an object-oriented program has to be able to declare unbound rule variables. A suitable approach in the context of dynamically typed object-oriented languages is to adapt the object-oriented language to interpret uninitialised temporary variables as unbound rule variables. A new kind of abstract

grammar element has to be introduced in the object-oriented language that represents these unbound rule variables.

The object-oriented program, from which an unbound query is activated, must be able to access the bindings of any unbound arguments of the query. A query binds all unbound arguments to objects, the only rule values in our model. One approach, again in the context of dynamically typed object-oriented languages, is that hybrid composition lets each unbound rule variable *become* the objects it is bound to.

In the case of a statically typed object-oriented language, the solution is not so transparent. A new class has to be defined to represent unbound rule variables. Whenever a rule variable is needed in the object-oriented program, this class needs to be explicitly instantiated. Moreover, a more explicit approach is required for accessing the bindings of a rule variable object since it cannot be so easily transformed into the objects it is bound to.

When manipulating bindings of a rule variable in an object-oriented program, one needs to take into account that rule-based languages do not distinguish between single or multiple bindings. The object-oriented program, on the other hand, treats a single object differently from a collection object. Hence, the correct kind of result needs to be returned. Again, distinct strategies are appropriate for statically and dynamically typed object-oriented languages. In the former, the inference has to be automatically converted to an object of the correct return type. This is straightforward unless multiple bindings are returned and a single object is expected. In this case, the only or first binding is returned. Once more, in dynamically typed languages, a more dynamic approach is warranted. A possible solution is representing the bindings as a special collection object, let us refer to this as a *dual single item collection*. This object behaves like a collection object or like a single object as required. In the second case, messages not understood by the dual single item collection are delegated to the first or only object in the bindings.

Activating Forward-Chaining Logic Rules from Objects

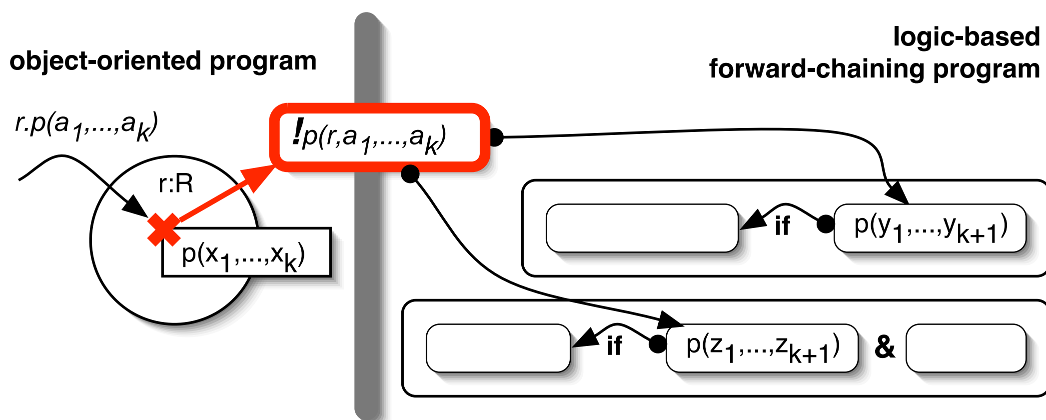


Figure 6.6: Activating forward-chaining logic rules from objects.

In a similar way, a method execution is interrupted and matched with a real assert, shown in figure 6.6. The assert results in the fact $p(r, a_1, ..., a_k)$ being asserted, denoted by the fat arrow pointing from the assert (in the red box) to the new fact. Once this fact is asserted, the forward chainer starts inferencing and triggers possibly multiple rules that

have a condition with predicate p and arity $k+1$. For example, in figure 6.6 two rules are triggered. The arguments $z_1, ..., z_{k+1}$ of the first condition of the lower rule are unified with the new fact's arguments $r, a_1, ..., a_k$.

Forward chainers do not have explicit inference results but side effects: new facts are inferred.

Sending Messages from Backward-Chaining Logic Rules

Figure 6.7 illustrates how a condition inference of a backward-chaining rule can result in a message being sent. On the left-hand side, two backward-chaining rules are shown. The condition of the upper rule, with predicate name m and arity k, can be inferred by rules that conclude the same predicate and arity, such as the lower rule. At this point, a matching message can be sent, shown in the red box in the figure. Once more, depending on the composition strategy, this message is sent before, after, or instead of the original condition inference. Moreover, candidate rules for inferring this condition might not exist at all, which simply results in the message being sent. The message selector is m, the original predicate's name. The receiver of the message corresponds to the binding of the first argument of the condition inference, x_1. Figure 6.7 shows the current set of bindings, $\{x_1 = a_1, ..., x_k = a_k\}$. In this case, the message receiver is a_1. The parameters of the message are the bindings of the remaining arguments of the condition inference, $x_2, ..., x_k$. For this example, the message is sent with parameters $a_2, ..., a_k$, which results in the method with name m, defined in the class R of a_1, being executed. The method's actual parameters are $a_2, ..., a_k$. When one of the arguments is unbound, the message cannot be sent and the condition inference fails. Note that multiple sets of bindings for a condition results in multiple condition inferences, one for each set of bindings.

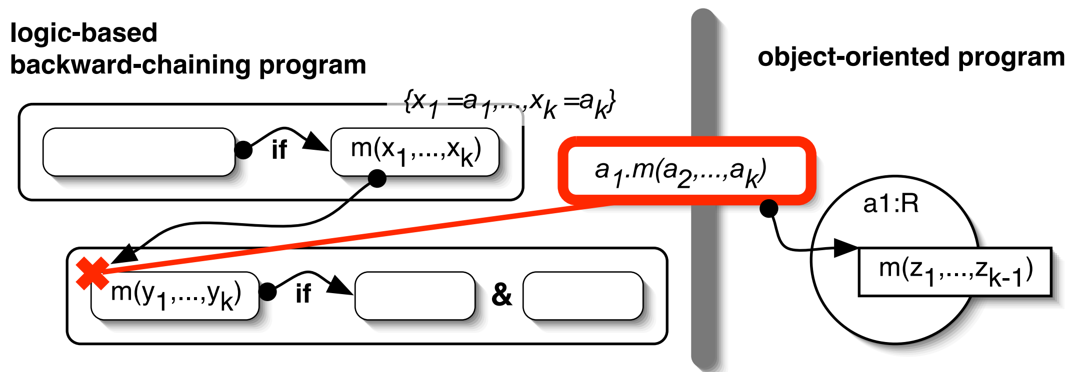


Figure 6.7: Sending messages from backward-chaining logic rules.

In the simplest case, a message send returns a boolean value which is interpreted as either a successful or failed condition inference. However, a typical result of inferring a condition in logic-based languages is that some unbound rule variables are bound to specific instances. When a message send at a condition inference join point returns a non-boolean value, it should also be possible to bind a variable in the condition to this value. However, in the normal matching illustrated earlier in figure 6.7, all k condition arguments are mapped to the receiver and $k-1$ parameters of a message send. In this case, all arguments have to be bound. In order to deal with non-boolean message send results, however, a match is needed between a condition predicate with k arguments and a message send with a receiver

and $k-1$ parameters. The remaining condition argument is then unified with the result of the message send. This is illustrated in figure 6.8. When the condition $m(x_1, ..., x_k)$ of the logic rule on the left-hand side is inferred, given the set of bindings shown in the figure, a message with selector m, receiver a_1 and parameters $a_2, ..., a_{k-1}$ can be sent. The result of this message send is unified with the last argument x_k of the condition.

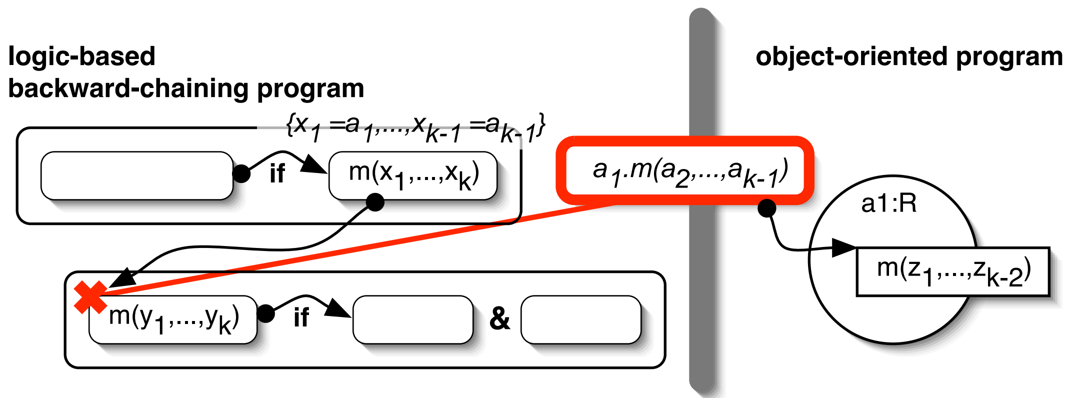


Figure 6.8: Returning message send results to logic rules.

Again, a similar question as before arises: how to decide which one of the matching messages is going to be sent, $a_1.m(a_2, ..., a_k)$ (figure 6.7) or $a_1.m(a_2, ..., a_{k-1})$ (figure 6.8). A possible strategy is determining which one of these two message sends actually corresponds to a real method, and execute this one. However, when methods exist for both message sends, unwanted composition possibly occurs. An alternative approach would be to extend the rule-based language with a unification operator similar to the match operator $=$ in production rule languages. This is illustrated in figure 6.9. The only difference with the situation in figure 6.8 is the condition of the rule: an explicit unification operator $=$ is used to alert hybrid composition that an explicit message send result is expected. The activation of the matching message send is unambiguous and achieved in the same way as before. This last approach, using the unification operator $=$, is the one we take and implement in chapter 7.

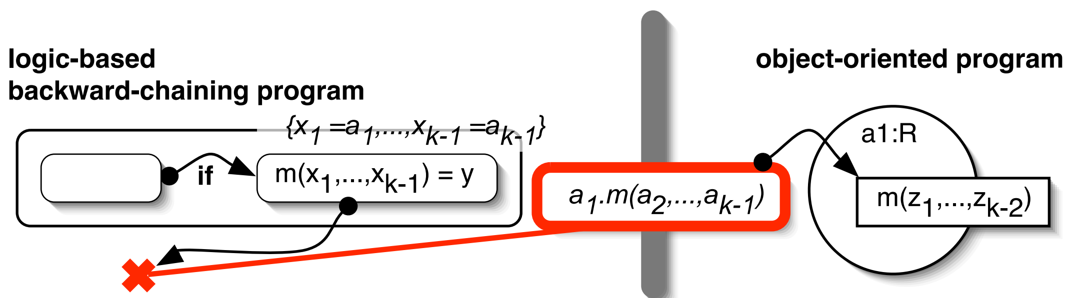


Figure 6.9: Returning message send results to logic rules using $=$.

Whether a rule-based language expects a single solution or multiple solutions for the rule-variable is irrelevant because they are treated in the same way during unification. However,

hybrid composition should be able to deduct if a collection of objects, which is returned as result of a message send, is a single collection object solution or multiple solutions. An approach for dealing with this is supporting the automatic conversion of instances of certain collection classes to multiple bindings for a rule variable.

Sending Messages from Forward-Chaining Logic Rules

In forward-chaining rules, messages can also be sent as a result of a conclusion generation, in addition to a condition inference. We review both situations, since inferring conditions is performed differently in forward-chaining logic languages. This is shown in figure 6.10.

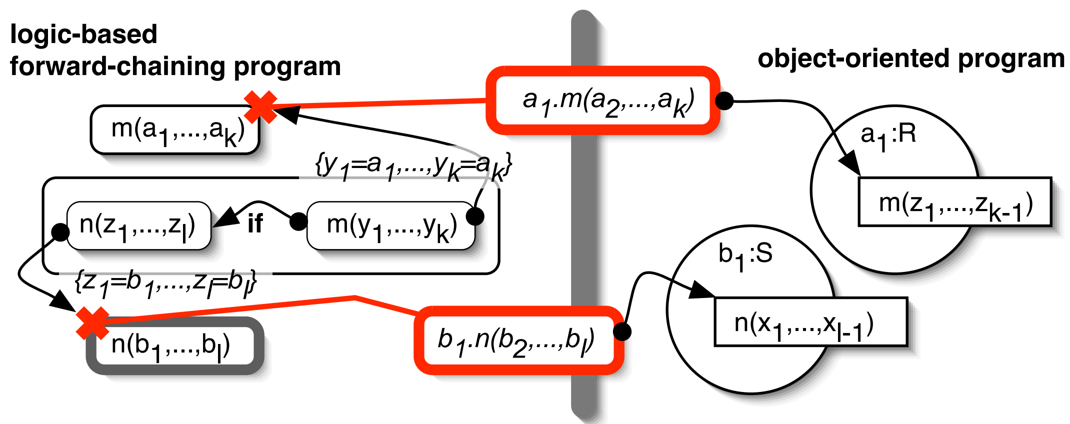


Figure 6.10: Sending messages from forward-chaining logic rules.

On the left-hand side, one forward-chaining rule is shown. The condition of this rule, with predicate name m and set of bindings $\{y_1 = a_1, ..., y_k = a_k\}$, can be inferred by facts that unify with it, such as $m(a_1, ..., a_k)$ shown in the figure. At this point, a matching message can be sent, denoted by the top red box. This message with selector m has receiver a_1 and parameters $a_2, ..., a_k$ and results in the corresponding method in object a_1, instance of R, being executed. Multiple sets of bindings for the condition result once more in multiple condition inferences and thus messages being sent. If the condition contains any unbound variables, the inference fails. The issues regarding returning message send results to logic rules are similar as for backward-chaining logic rules.

When the rule in figure 6.10 concludes the predicate named n with arguments $z_1, ..., z_l$, a fact is typically generated for each set of bindings for this predicate. The figure shows one set of bindings, $\{z_1 = b_1, ..., z_l = b_l\}$ and the corresponding new fact that can be generated, $n(b_1, ..., b_l)$. At this point, a matching message can be sent, shown by the bottom red box, taking the same approach as before. Since a conclusion does not expect any kind of result, any results of message sends are ignored.

6.6.3 Linguistic Symbiosis with Production Rule Languages

This section discusses activating rules from objects and sending messages from rules for both backward-chaining and forward-chaining production rule languages. The discussion also remains independent of actual production rule languages.

Again, we have implemented the linguistic symbiosis for Smalltalk and a new generation forward-chaining production rule language. In the next chapter, we provide examples of

this flavour of the linguistic symbiosis in section 7.2.3. That way, the reader can refer to the concrete examples in this section when reading the general and language-independent setup of the linguistic symbiosis in the following. Note that we do not implement the linguistic symbiosis for backward-chaining production rule languages. The reason is that this last flavour can be reconstructed from the linguistic symbiosis with backward-chaining logic languages and forward-chaining production rule languages.

Activating Backward-Chaining Production Rules from Objects

Instead of activating a production rule query at normal object-oriented join points, i.e. method executions, it can be activated when an attribute of an object is accessed. This is shown in figure 6.11.

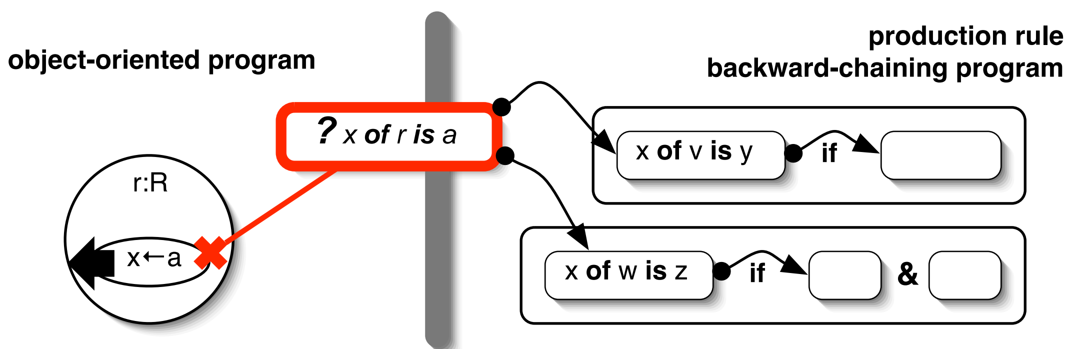


Figure 6.11: Activating backward-chaining production rules from objects.

When the attribute x of object r is accessed, a matching query can be activated. In figure 6.11, an attribute is denoted by the oval inside object r. Accessing this attribute is depicted by the black arrow leaving the attribute. The query, shown in the red box, is qualified with the attribute name x. The object r is an argument of the query as well as the value of the attribute a, if it is known. This query triggers the two rules on the right-hand side which conclude values for the attribute x of objects. Note that we do not require to extend the object-oriented language with a representation for unbound rule variables in order to activate unbound queries. Indeed, backward-chaining production rule languages simply interpret an uninitialised object attribute as an unbound variable.

The result of a query is typically an indication of success or failure, and optionally provides, in case of success, bindings for unbound variables in the query. When a production rule query results in an attribute x of an object of class R being inferred, hybrid composition simply sets the attribute with the new value.

The issue of returning multiple or single bindings to an object-oriented program is also relevant in the context of linguistic symbiosis with production rule languages. The same discussion as for linguistic symbiosis with a logic-based language (section 6.6.2) applies here.

Activating Forward-Chaining Production Rules from Objects

Forward-chaining production rules are also not activated at the normal object-oriented join points, but instead when objects are created or attributes of objects are set. This is illustrated in figure 6.12.

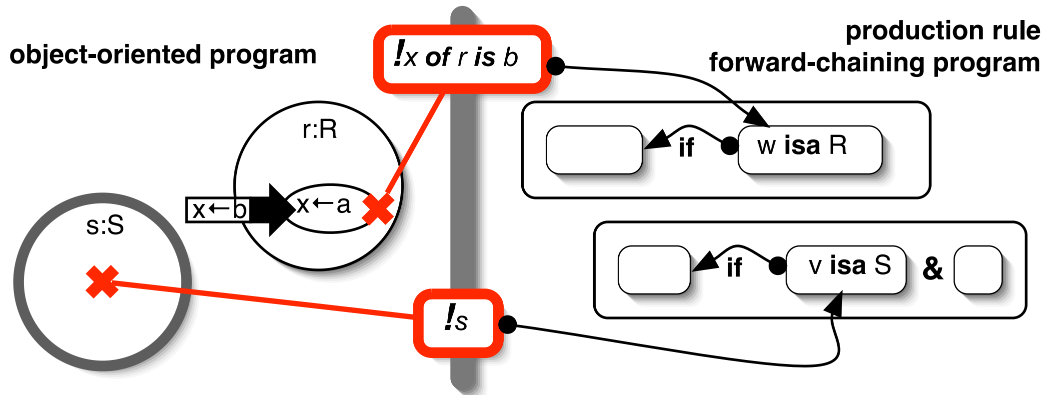


Figure 6.12: Activating forward-chaining production rules from objects.

In the first case, the class S is instantiated, depicted by the bottom left object s in the figure. At this point, this object is asserted in the working memory of the production system. This triggers the bottom rule because it matches objects of class S.

In the second case, the attribute x of the object r is set with a new value b. This is shown in figure 6.12 by the black arrow pointing to the field x, preceded by the assignment $x \leftarrow b$. At this point, an assert x **of** r **is** b can be activated. Again, the attribute name x makes up the qualifier of the assert while r and b are the arguments of the assert. This triggers the top rule because it matches objects of class R.

Forward chainers do not have explicit inference results but side effects: new objects or changes in objects are inferred.

Sending Messages from Backward-Chaining Production Rules

Since we only consider production rule languages that have been extended with object-oriented features, as found in most existing hybrid production systems, matching production rule join points with object-oriented behaviour activation is trivial: each condition and conclusion message send statement implemented in the production rule language is in fact evaluated in the object-oriented language. Since this is exactly what happens in existing hybrid production systems, we do not contribute to this side of the integration. Therefore, the only purpose of the ensuing explanation is to put this step in the same perspective as the previous ones.

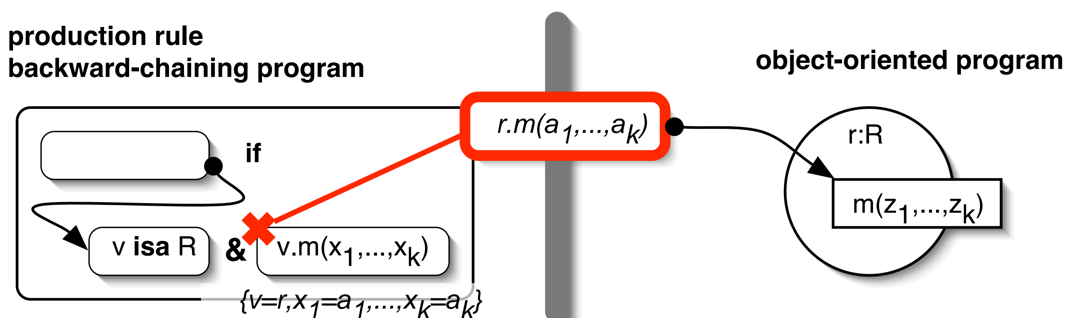


Figure 6.13: Sending messages from backward-chaining production rules.

Figure 6.13 illustrates how the condition message send in the production rule on the left with receiver v, selector m and arguments $x_1, ..., x_k$ is matched to a message send with selector m for each set of bindings. Considering the set of bindings in the figure, $\{v = r, x_1 = a_1, ..., x_k = a_k\}$, a message m is sent to receiver r with parameters $a_1, ..., a_k$, shown in the red box.

The result of a message send needs to be returned to the production rules. In the simplest case, a message send returns a boolean value which is interpreted as either a successful or failed condition inference. In case of a more sophisticated interaction, the rules require another value as the result of a message send. Typically, hybrid production systems provide an explicit operator that matches message send results with a rule variable. For example, consider the production rule on the left-hand side in figure 6.14. As explained earlier, the condition message send $v.m(x_1, ..., x_k)$ results in the message with selector m being sent for each set of bindings of the condition. In the example this results in $r.m(a_1, ..., a_k)$ being sent, depicted in the red box. As this is a message send that returns a non-boolean result, the result is matched with y in the condition using the matching operator $=$.

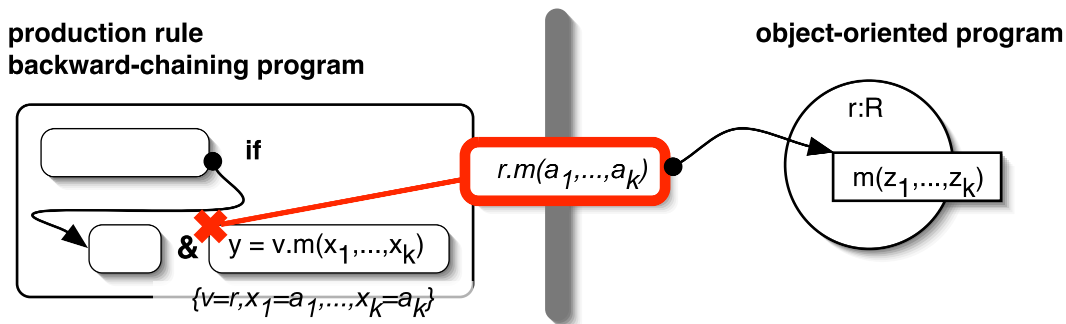


Figure 6.14: Returning message send results to production rules.

Sending Messages from Forward-Chaining Production Rules

In the case of forward-chaining production rules, the condition inference join points are matched to message sends in exactly the same manner as in backward-chaining production rules. Furthermore, results of message sends are also taken care of in the same way. Figure 6.15 illustrates this in the bottom red box. This figure also depicts a conclusion generation join point. The conclusion message send to receiver w, with selector n and arguments $x_1, ..., y_l$ results in the matching message $s.n(b_1, ..., b_l)$ being sent, in the top red box. The matching is performed entirely similarly to condition inference join points in production rules. However, a conclusion does not expect any kind of result, so any results of conclusion message sends are ignored.

6.7 Hybrid Aspects

The previous section presents hybrid composition, but also reveals that this first part of our approach does not achieve full obliviousness: the qualifier of a behaviour activation has to match exactly with the qualifier of a join point in order for composition to occur. Moreover, more sophisticated exchange of values during behaviour activation requires adding specific features to the base languages. However, the advantage of hybrid composition is its universal and automatic integration of rule-based and object-oriented programs.

Figure 6.15: Sending messages from forward-chaining production rules.

The second, most important part of our approach consists of hybrid aspects. A hybrid aspect has the classical pointcut/advice structure which allows it to specify join points and execute an advice program at each join point. Hybrid aspects employ the same join point model as hybrid composition, safe for the object instantiation and attribute access and mutation join points used for activating production rules. The advice program can consist of different behaviour activation statements, which do not have to correspond and match with the specified join points. Furthermore, the specific language features are no longer required in the base languages themselves, as is the case with hybrid composition, but are supported by the hybrid aspect language. This results in base languages that are oblivious to being integrated with each other. As such, part of hybrid composition lays the groundwork for hybrid aspects.

Hybrid aspects are presented in the remainder of this section, in a format different from the one used for hybrid composition. We present an abstract grammar of a hybrid aspect language for object-oriented and rule-based base languages. Again, we consider the base languages described in section 6.5, which means that once more, we take different rule-based formalisms and chaining strategies into account. The abstract grammar is for the most part common for all rule-based languages, although some differences occur due to the aforementioned variability.

The disadvantage of using the abstract grammar as a medium for explaining a language, is that it lacks a *concrete* grammar. Hence, it is often difficult to imagine how the language is actually used and what programs written in this language look like. Therefore, we refer the reader again to the corresponding sections in the next chapter, which provide example hybrid aspects for different kinds of rule-based languages. Indeed, we have implemented hybrid aspects for Smalltalk and a Prolog-like language, Smalltalk and a forward-chaining logic language, and Smalltalk and a forward-chaining production rule language. Example hybrid aspects for these base languages are provided in section 7.3.1, section 7.3.2 and section 7.3.3, respectively.

Hybrid aspects follow the basic pointcut/advice structure of aspect-based approaches. This is represented in our abstract grammar as follows:

```
<hybrid aspect definition> ::=
   ASP(<pointcut definition> , <advice definition>)
```

which means that a hybrid aspect definition is identified with the tag ASP and consists of two elements, more specifically a hybrid pointcut definition and a hybrid advice definition.

Note that we use a postfix notation: the tag precedes a list of abstract grammar elements, which are enclosed between brackets '(' and ')', and separated by commas ','.

In hybrid aspect definitions, the pointcut definition, which specifies the context in which the aspect should be applied, is tightly coupled with the advice definition, which represents the alternative or additional behaviour to be activated in the context. This results in hybrid aspects that depend heavily on the context in which they are applied and thus are hard to reuse in other contexts. Many aspect-oriented approaches exist that improve this original pointcut/advice approach in order to obtain reusable aspects and composition of aspects, such as *Aspectual Components* [Lieberherr et al., 1999], *Aspectual Collaborations* [Lieberherr et al., 2003], *Caesar* [Mezini & Ostermann, 2003] and JAsCo [Suvée et al., 2003] (also presented in chapter 3). As mentioned before, we only consider basic aspect-oriented features in our work because this is a first foray into the application of aspect-oriented programming to hybrid systems. In the last chapter, however, we discuss how our hybrid aspects can be rendered more reusable by extending our model in non-fundamental ways in future work (section 9.2).

Note that we use the terminology introduced by AspectJ. One of the reasons is that it is one of the most popular pointcut/advice-based approaches. Another reason is that hybrid aspects conform to a basic pointcut/advice structure, which makes them very similar — in structure at least — to AspectJ. Hybrid aspects do not provide all the features AspectJ supports but extends a basic subset for different kinds of rule-based languages.

The remainder of this section follows the structure of the abstract grammar: significant program elements are discussed one by one. The abstract grammar is shown in its entirety in appendix B.

6.7.1 Hybrid Pointcut Definitions

A pointcut denotes a set of join points, and optionally, some values from the execution context of those join points. In our context, join points in both the object-oriented and rule-based programs can be denoted, hence the term hybrid pointcut definitions. Note that we refer to the part of the aspect language that deals with defining pointcuts as the *pointcut language*. Since many existing aspect-oriented programming approaches have a very powerful and sophisticated pointcut language, this dissertation does not focus on nor contribute to this area. Our work primarily focuses on determining join points for both object-oriented and rule-based base languages, presented earlier in sections 6.5.1 and 6.5.2, and on hybrid advice. The latter is not concerned with how the required join points are denoted by the pointcut language, only that they are. As we explain later on, hybrid advice is parametrised with values from the interrupted execution context and executed at each of the join points. One could say that we regard the pointcut language as some kind of black box: it considers a certain join point model and pointcuts defined in it denote the concrete join points.

Usually, pointcuts can be combined using and, or and not operators, as represented below in our abstract grammar:

```
<pointcut definition> ::=
  AND(<pointcut definition>+) |
  OR (<pointcut definition>+) |
  NOT(<pointcut definition>)  |
  <pointcut designator>
```

The atomic elements in these conjunctions, disjunctions and negations are pointcut designators. These are described next.

6.7.2 Pointcut Designators

A pointcut definition describes a pointcut using pointcut designators, which state some property about join points. Join points that match a designator are picked out. There are different kinds of primitive pointcut designators typically provided by existing pointcut languages. Some of these target a particular kind of join point. Therefore, there generally exists exactly one primitive pointcut designator for each kind of join point. In our case, this results in a pointcut designator for method executions, which are the only kind of object-oriented join point we consider, and one pointcut designator for each kind of rule-based join point, more specifically condition inference and conclusion generation. This is reflected in our abstract grammar for pointcut designators below:

```
<pointcut designator> ::=
   <method execution pointcut designator> |
   <condition inference pointcut designator> |
   <conclusion generation pointcut designator>
```

Since conclusion generation join points only occur in forward-chaining rule-based languages, this element is italicised. This variability in our abstract grammar actually means that two different hybrid aspect languages are described: one for forward-chaining and one for backward-chaining rule-based languages. Of course, the entire non-italicised part of our abstract grammar is common for both types of rule-based languages.

Existing aspect-oriented programming approaches generally provide other primitive pointcut designators, targeting any kind of join point at which a certain property holds. An example is AspectJ's *instanceof(<classname>)* pointcut designator which targets all kinds of join points — method call, reception and execution — where the object that was the receiver of the message that activated the method is an instance of a class or subclass named *<classname>*. Another example is the more sophisticated pointcut designator *cflow(<joinpoint>)*, also defined in AspectJ, which allows picking out join points based on whether they are in a particular control flow relationship with another join point *<joinpoint>)*. Both pointcut designators are introduced in chapter 3. We do not consider these alternative pointcut designators in our approach, but discuss in the last chapter how they can be added as part of future work (section 9.2). Important to remember is that the underlying join point model remains independent of whatever pointcut designators are introduced (such as cflow) and whatever qualification of join points is used in pointcut designators.

Pointcut designators are parametrised with qualifiers in order to distinguish the desired join points from the others. In the case of the first kind of pointcut designators, the ones that target a particular kind of join point, qualifiers of join points are used as parameter. The more elaborate the qualifiers used by the pointcut designators are, the more (run-time) properties can be used to distinguish between concrete join points. However, earlier in this dissertation we state that we use basic qualification of join points.

Since some qualifiers are identifiers, such as method names, most existing aspect-oriented approaches support wildcarding of these identifiers in order to pinpoint many heterogeneous join points in one pointcut designator. Again, we do not support this feature, yet obviously our approach can be extended with it in a straightforward manner.

Some existing aspect-oriented approaches, such as AspectJ and *Andrew* [Gybels & Brichau, 2003], allow pointcut designators to contain variables and provide a mechanism for binding a particular value of a join point to the variable. The advice that corresponds to pointcuts consisting of pointcut designators with variables, is parametrised with the values to which the variables are bound. This is a very clean approach to capturing values from the interrupted execution context at a join point and exposing them to advice. Alternatively, the entire join point context is made available to the advice. AspectJ supports this second approach in addition to the first, by providing special variables to be used in the advice body, such as *thisjoinpoint*, which are bound to an object representing the current join point. *AspectS* also supports this second approach, but parametrises the advice with one parameter whose actual value is an object representing the current join point [Hirschfeld, 2002]. Both AspectJ and AspectS provide reflective access to the join point object in order to retrieve information from it, for example, actual parameters of an interrupted method execution. However, this tends to clutter up the advice body with statements that retrieve the required values from the join point object, whereas the former approach passes them to the advice. Our hybrid aspects follow the second approach, but as mentioned earlier, a more sophisticated pointcut language can replace the more limited one presented here. This does not require changes to our join point model and only limited adaptation to our hybrid advice: instead of being parametrised with one context object it is parametrised with captured values from the context.

Object-Oriented Pointcut Designators

Since our object-oriented qualifiers consist of method and class name (section 6.5.1), our only object-oriented pointcut designator captures all executions of a method with a certain name which is defined in or inherited by a certain class. This is expressed in the part of our abstract grammar below:

```
<method execution pointcut designator> ::=
   MTH(<class name> , <method name>)

<class name> ::= <identifier>

<method name> ::= <identifier>
```

As the above discussion pointed out, if a pointcut designator includes more information about a method execution, such as number of parameters, type of the actual parameters and return value, more flexible specification of join points is possible. This can be achieved by extending our pointcut designator to include this extra information and taking this information into account when matching join points against concrete pointcut designators.

Rule-Based Pointcut Designators

Typically, there exists a rule-based pointcut designator for each kind of rule-based join point: condition inferences and conclusion generations. As mentioned in section 6.5.2, rule-based qualifiers depend on the formalism on which the rule-based language is based.

Logic-Based Pointcut Designators rely on logic-based qualifiers, which in our model consist of predicate name and arity. The abstract grammar elements for these designators are given below:

```
<condition inference pointcut designator> ::=
   CND(<predicate name> , <predicate arity>)

<conclusion generation pointcut designator> ::=
   CNC(<predicate name> , <predicate arity>)

<predicate name> ::= <identifier>

<predicate arity> ::= <number>
```

Remember that the second pointcut designator is only meaningful when having a logic-based forward chainer as base language, which is why it is italicised.

Production Rule Pointcut Designators employ production rule qualifiers. Remember that these qualifiers are similar to object-oriented qualifiers because the kind of production rule language we consider incorporates message send statements in conditions and, if it is a forward chainer, in conclusions as well. The abstract grammar elements for production rule pointcut designators are given below:

```
<condition inference pointcut designator> ::=
   CND(<class name> , <method name>)

<conclusion generation pointcut designator> ::=
   CNC(<class name> , <method name>)

<class name> ::= <identifier>

<method name> ::= <identifier>
```

Once more, the second pointcut designator only appears in hybrid aspect languages with forward-chaining production rule languages as base language.

Note that the above rule-based pointcut designators produce partly distinct hybrid aspect languages for rule-based languages that differ with respect to formalism and chaining strategy.

6.7.3 Hybrid Advice

The ingredients of hybrid advice are elucidated by the part of our abstract grammar shown below:

```
<advice definition> ::=
   ADV(<aspin variables> , <advice argument> , <temporary variables>
       , <advice statements>)
```

First of all, hybrid aspects can be instantiated and each aspect instance, also referred to as an *aspin*, encapsulates state in aspin variables. Furthermore, as discussed in section 6.7.2, hybrid advice has the entire join point context as single argument. Thirdly, hybrid advice is able to declare temporary variables. Finally, the last element is the body of hybrid advice, which consists of a list of advice statements. These four elements are discussed in the following.

6.7.4 Aspin Variables

As supported by most aspect-oriented approaches based on pointcut/advice, aspects can be instantiated and encapsulate state. The state consists of a list of aspin variables, which are in turn identifiers, expressed as follows in our abstract grammar:

```
<aspin variables> ::= LST(<variable>*)

<variable> ::= <identifier>
```

In our context, aspin variables are necessary for capturing a value at a certain join point with an aspect and passing this value to another aspect which employs it to integrate some alternative behaviour. The first aspect "passes" the value by setting one of the second aspect's aspin variables with this value. This is one of the typical uses of aspects when integrating object-oriented and rule-based programs, illustrated in chapter 2 (section) and [Cibrán et al., 2003a].

Aspin variables can refer to objects or rule variables, which are either bound to objects or not bound at all.

6.7.5 Advice Argument

A hybrid aspect's advice is parametrised with the execution context which is interrupted at the join points denoted by the aspect's pointcut. Hence, the advice argument is a variable:

```
<advice argument> ::= <variable>
```

When hybrid advice would have specific values from join points as arguments instead of the entire join point context, the advice argument becomes a list of variables.

A challenge introduced by the use of a rule-based as well as an object-oriented base language, is that join points can be rule-based or object-oriented. An interrupted object-oriented execution context contains receiver and actual parameters, which are objects. An interrupted rule-based execution context contains predicate arguments, which are either rule values or rule variables, and the current set of bindings. The set of bindings contains zero or one binding for each rule variable. The rule-based languages we consider use objects as rule values and are able to bind rule variables to objects (section 6.5.2). So in general, rule-based contexts contain objects and rule variables which are or are not bound to objects. There are some exceptions, however: conclusion generation contexts do not contain unbound rule variables, and production rule contexts, either condition inference or conclusion generation, never contain unbound rule variables (section 6.5.2).

Another issue which results from having rule-based and object-oriented join points, is how their contexts are going to be represented and manipulated in the hybrid advice body. In our approach, as in AspectJ and AspectS discussed in section 6.7.2, the interrupted object-oriented execution context is represented by an object and reflective access is provided in order to retrieve information from it. The reflective access is enabled by sending messages to the context objects, which returns objects such as receiver and actual parameters. Our hybrid aspect language also represents rule-based join point contexts as objects. Such an object contains the arguments of the interrupted condition or conclusion as well as the current set of bindings. Retrieving the arguments from such a context object using the reflective access, results in objects or rule variables being returned. The value to which

a rule variable is bound can be looked up in the set of bindings of the context object, also achieved by using the reflective access.

Hybrid advice has sets of bindings, which is initially the set of bindings from the rule-based context or an empty set of binding if the context is object-oriented. The reason is that rule variables can be created ex nihilo and bound to some object(s) in the advice body. This requires the new rule variable and its bindings to be added to the advice's set of bindings.

6.7.6 Temporary Variables

Hybrid advice supports the declaration of temporary variables. The corresponding abstract grammar element consists of a list of variables:

```
<temporary variables> ::= LST(<variable>*)
```

Temporary variables have a dual role. First of all, they are used for storing values from the interrupted execution context, introduced in section 6.7.5. As such, after initialisation, temporary variables refer to objects or rule variables, which can be bound or unbound. A second use of temporary variables is for the ex nihilo creation of rule variables. When an uninitialised temporary variable is used in declarative expressions, which are presented later on, it is interpreted to be an unbound rule variable.

6.7.7 Advice Statements

An advice body consists of a list of advice statements:

```
<advice statements> ::= LST(<advice statement>*)
```

An important question to address is what the execution model of a hybrid advice program is, or, in other words, what the programming paradigm of hybrid advice is. Since our hybrid aspect language combines features and values of an object-oriented and a rule-based language, it should adhere to both paradigms and provide a combined execution model.

The object-oriented programming paradigm allows us to view a program execution as dialogueing objects: objects encapsulate state and behaviour and behaviour is activated by one object sending a message to another. However, at the method level, the execution model is more imperative: a method body is regarded as a list of instructions that are executed in a particular order.

Conversely, rule-based programs are made up of rules, which consist of a conjunction of conditions from which, if they are true, the rule's conclusion can be inferred. Theoretically, one should only keep the declarative semantics in mind when defining rules, yet it is often impossible to ignore the procedural semantics. Moreover, production rule languages are not very declarative to start with since they specify in which situation, expressed by conditions, an action has to be performed, defined in the conclusion. Although each rule-based language has a different execution model, they all adhere more or less to the following procedural semantics when evaluating a rule: first try to satisfy the first condition, then, if successful, try to satisfy the second condition, and so on until all conditions are established, after which we can conclude that the conclusion is true. A crucial difference with the imperative execution model described earlier, is that satisfying a condition can result in multiple bindings for a rule variable which necessitates *branching* the evaluation of the remaining conditions for each binding. Particular rule-based languages provide different strategies for visiting the nodes in the resulting tree, for example in a depth-first or breadth-first manner.

Our hybrid aspect language also provides expressions that can cause branching of the evaluation of the ensuing statements. Indeed, as we explain later on, a query can bind a rule variable to multiple objects and a rule variable can be unified or matched with multiple objects. As such, the execution model of a hybrid advice body is similar to the one for rule-based languages described above: the list of advice statements should be regarded as a conjunction of conditions rather than a sequence of instructions. Additionally, disjunction and negation of conditions are provided, which are evaluated as in the rule-based language. This is represented in our abstract grammar as follows:

```
<advice statement> ::=
   OR (<advice statement>+) |
   NOT(<advice statement>)  |
   <expression>
```

Each statement is evaluated with the current set of bindings. Each such evaluation indicates success or failure, which determines if the next statement is evaluated. The expressions can be divided in two categories: "imperative" and "declarative" expressions:

```
<expression> ::=
   <imperative expression> |
   <declarative expression>
```

The first are the typical object-oriented expressions such as variables, assignments and message sends, as well as proceeding with the original, object-oriented execution:

```
<imperative expression> ::=
   <variable> |
   <object-oriented proceed> |
   <assignment> |
   <message send>
```

Imperative expressions such as variables and assignment always indicate success so that the next statement can be evaluated in a sequential manner. However, message sends and object-oriented proceed expressions indicate success if they evaluate to the boolean object true or failure if they evaluate to false. Imperative expressions cannot cause branching of the evaluation of the remaining statements because they are not able to bind rule variables, let alone to multiple bindings. Imperative expressions evaluate to a concrete value, more specifically an object or a rule variable. Therefore, these expressions can be nested in other expressions. Note the distinction between indicating success or failure, which is used by the basic steps of our execution mode, and evaluating to a value, which is used when an imperative expression is nested in another expression. An object is used as is in an imperative expression, whereas a rule variable's binding is looked up and passed to the expression. Unbound rule variables cannot be used in imperative expressions. The different expressions are discussed in more detail later on.

Declarative expressions consists of rule-based expressions such as unification/match, queries and asserts, as well as proceeding with the original, rule-based execution:

```
<declarative expression> ::=
   <rule-based proceed> |
   <unification/match> |
   <query> |
   <assert>
```

All declarative expressions indicate success or failure, and all, safe for asserts, can bind rule variables. When declarative expressions indicate a failure, evaluation of the remaining statements is aborted and the next branch is evaluated with its set of bindings. On the other hand, when a declarative expression indicates success, the next statement is evaluated. Additionally, when sets of bindings are inferred for unbound rule variables, the evaluation of the next statement is branched for each set of bindings. Arguments of declarative expressions are imperative expressions, which evaluate to objects or rule variables.

The evaluation result of the last statement in hybrid advice is returned to the interrupted execution context. When the latter is object-oriented, an object is expected. Imperative expressions evaluate to objects, but declarative expressions indicate success or failure. This is converted to the corresponding boolean object and returned to the object-oriented context. When the context is rule-based, typically an indication of success or failure is expected or, in the case of conclusion generation join points, nothing at all is expected. Declarative expressions indicate success or failure, but imperative expressions do not. When the last statement is an imperative expression, it should evaluate to a boolean object which is converted to an indication of success or failure. Sometimes a value is expected by the interrupted rule-based context, more specifically in a condition inference which contains an explicit unification or match operator. In this case, the last statement should evaluate to an object or rule variable which is explicitly returned to the interrupted rule-based context.

6.7.8 Rule-Based and Object-Oriented Proceeds

In our approach we only consider one kind of advice, typically referred to as *around advice*. This kind of advice replaces the original execution that is interrupted and optionally, allows the original execution to proceed using a proceed expression. Around advice is able to express the other typical kinds of advice that execute before, after or instead of the original execution by placing the proceed at the end, in the beginning or not at all in the hybrid advice body.

Since we allow both rule-based and object-oriented programs to be interrupted by hybrid aspects, we have to consider both rule-based and object-oriented proceed expressions, which are represented in a similar way but evaluated differently:

```
<rule-based proceed> ::= RPR()

<object-oriented proceed> ::= OPR()
```

A proceed expression continues with the interrupted execution which means that a rule-based proceed is a declarative expression, while an object-oriented proceed is imperative. The former evaluates to an indication of success or failure. A special case are conclusion generations, which always evaluate to an indication of success. An object-oriented execution evaluates to an object, which means that an object-oriented proceed can be nested in other expressions.

6.7.9 Assignment

Our hybrid aspect language provides an assignment statement for initialising or changing aspin variables and temporary variables. The relevant part of our abstract grammar is given below:

```
<assignment> ::= ASS(<variable> , <imperative expression>)
```

The left-hand side is a variable to which the result of evaluating the right-hand side, an imperative expression, is assigned. In hybrid advice, the left-hand side has to be an aspin variable or a temporary variable because it is not allowed to assign a value to the only other kind of variable, i.e. the advice argument. The imperative expression evaluates to a value, which is an object or a rule variable. The assignment itself operates as in any imperative programming language: the reference of the variable on the left-hand side is replaced with a reference to the value to which the right-hand side evaluates. This means that rule variables themselves are assigned to variables in the left-hand side, not the objects they might be bound to.

Evaluating an assignment returns the value to which its right-hand side evaluates.

6.7.10 Unification or Match

Since hybrid advice contains aspin and temporary variables that may refer to rule variables, it needs to be outfitted with an expression for binding these rule variables to objects. The abstract grammar element representing this expression is shown below:

```
<unification/match> ::= UNI(<variable> , <imperative expression>)
```

This element is referred to as unification/matching because in logic-based languages the first is employed whereas production rule languages use the second. Nevertheless, their basic structure and outcome is similar.

As in assignment expressions, the variable in the left-hand side of a unification cannot be the advice argument, since in our approach this is an object representing the entire join point context. However, if our approach is extended with more sophisticated pointcut definitions which capture values from a join point context and expose them to the advice, then the advice is instead parametrised with these values (review the discussion in section 6.7.2). As a result, when executing the hybrid advice, its arguments are bound to the concrete values from the join point context which are captured by the pointcut. In this case, it is valid to use these arguments, if they are rule variables, in the left-hand side of a unification or match expression.

Also similar to the assignment expression is the imperative expression on the right-hand side of a unification or match, which evaluates to an object, or a bound or unbound rule variable.

Unification and match operate in exactly the same way as provided by modern-day rule-based languages that consider objects as rule values and bind rule variables to them. They return an indication of success or failure. The unification/match expression has an impact on the current set of bindings: a new binding for a previously unbound rule variable can be added to it. Similar to our approach in hybrid composition, however, unifying or matching collection objects results in each element of the collection being unified or matched separately with the left-hand side.

6.7.11 Message Send

A message send consists of a receiver, a message selector — which is the same as a method name — and a list of parameters. This is expressed in the following elements of our abstract grammar:

```
<message send> ::=
   MSG(<receiver> , <method name> , <imperative expressions>)
```

```
<receiver> ::= <imperative expression>
```

```
<imperative expressions> ::= LST(<imperative expression>*)
```

Both the receiver and the parameters of a message send are imperative expressions. These expressions are evaluated first before the message is actually sent. Imperative expressions evaluate to an object or a rule variable, which is looked up in the current set of bindings. If the rule variable is bound to an object, the message send is evaluated with the object. If the rule variable is unbound, the message send is erroneous. The result of a message send is an object. However, when a message is sent to the object representing the context with which the advice is parametrised (section 6.7.5), the return value can also be a rule variable.

6.7.12 Query

As in hybrid composition, queries are activated differently for logic-based languages and production rule languages. Nevertheless, the arguments of a query, whether it is based on logic or on production rules, are imperative expressions, which evaluate to an an object or a rule variable. After evaluation, a query gives an indication of success or failure. If a rule variable is unbound and the query is successful, bindings for the rule variable have been inferred. As with most declarative expressions, a query may cause branching of the evaluation of the remaining statements for each set of bindings of the previously unbound rule variables.

Logic-Based Query

A logic-based query consists of a predicate name and a list of arguments, as expressed below:

```
<query> ::=
   QRY(<predicate name> , <imperative expressions>)
```

Production Rule Query

A production rule query consists of an attribute name and arguments that represent the object whose attribute is being inferred, and the attribute itself.

```
<query> ::=
   QRY(<imperative expression> , <attribute name> ,
       <imperative expression>)
```

6.7.13 Assert

Similar to query expressions, assert expressions differ for logic-based and production rule languages, and the arguments of an assert are always imperative expressions. The difference with queries is that the imperative expressions should evaluate to an object or a bound rule variable. In the case of the latter, the binding is looked up in the current set of bindings and used as argument of the assert. Note that, in principle, logic-based asserts can contain unbound rule variables, although we do not consider such languages in our model.

Logic-Based Assert

A logic-based assert consists of a predicate name and a list of arguments, as expressed below:

```
<assert> ::=
   AST(<predicate name> , <imperative expressions>)
```

Production Rule Assert

A production rule assert either has one argument, which is the object being asserted, or an attribute name and two arguments, which represent the object whose attribute is being asserted and the attribute itself.

```
<assert> ::=
   AST(<imperative expression>) |
   AST(<imperative expression> , <attribute name> ,
       <imperative expression>)
```

6.8 Conclusion

This section draws conclusions about hybrid composition and hybrid aspects. We discuss several issues.

6.8.1 Integration Philosophy

The integration philosophy clearly has an impact on hybrid composition and hybrid aspects. We discuss the two identified integration philosophies briefly in the conclusion of chapter 5 on hybrid systems.

One philosophy is to adapt at least one of the integrated languages to be more compatible with the other. Most modern-day hybrid production systems follow this approach and extend the production rule language with object-oriented features such as message sending. This integration philosophy leads to a very straightforward strategy for hybrid composition: a message send in a production rule is always evaluated in the object-oriented language. It also simplifies hybrid aspects because unification or matching of message send results are already incorporated in the production rule language. The drawback of this philosophy is that it breaks obliviousness at the language level right from the beginning since the object-oriented paradigm is made explicit in the production rule language. At this point, neither hybrid composition nor hybrid aspects can counter the loss of obliviousness but only adapt to it.

This contrasts with the other integration philosophy, the one employed by most logic-based hybrid systems. In such systems, both languages only exhibit features that are characteristic for their paradigms. Integration is achieved because each language is outfitted with an interface to the other language, expressed in its own language features. Special keywords or characters notify the evaluator of a dispatch to the other language. This philosophy also breaks obliviousness at the language level, but this can be (partly) remedied by hybrid composition. It uses an alternative strategy: linguistic symbiosis, which determines an unambiguous match between elements of both languages based on their qualifiers. This is used instead of special keywords or characters to notify the language of a dispatch to the other language. As a result, each language still uses its typical language features for

exchanging values with the other language, but the "interface" has become entirely transparent. Hybrid aspects even maintain language obliviousness with respect to the issues related to value exchange. In the evaluation chapter (chapter 8), we discuss, among others, the exact effects of hybrid composition and hybrid aspects on obliviousness at the language level.

6.8.2 Rule-Based Formalisms and Chaining Strategies

Different flavours of hybrid composition need to be conceived for different rule-based formalisms and chaining strategies. However, there are commonalities between the flavours for logic-based languages that are independent of the chaining strategy: the same qualifiers are used which results in a similar one-to-one mapping between join points and behaviour activation, whether it is a query or an assert. Secondly, similarities between the rule-based languages that employ the same chaining strategies are apparent. Languages based on different formalisms but with the same chaining strategy have the same general correspondence between join points and behaviour activation: object-oriented join points either correspond to an assert or to a query. Moreover, the same language integration issues are relevant.

Our hybrid aspect language is for the most part the same for different rule-based languages. The abstract grammar has slightly different elements for rule-based languages that differ in formalism and/or chaining strategy. However, the general execution model for hybrid advice is the same for all rule-based languages. Moreover, the same values are considered in hybrid aspects: objects or rule variables, which are either bound or unbound.

Chapter 7

Hybrid Composition and Hybrid Aspects in OReA

In this chapter we present proof-of-concept implementations of hybrid composition and hybrid aspects for integrating several concrete rule-based languages and the object-oriented language Smalltalk. The next section presents our atelier in which the implementations are developed. Then, section 7.2 presents implementations of hybrid composition for three rule-based languages. Section 7.3 does the same for hybrid aspects. As such, this chapter provides simple, yet concrete examples of rule-based and object-oriented programs that are integrated using hybrid composition and hybrid aspects. The next chapter presents a more complex example and uses it to evaluate our approach.

7.1 The Atelier OReA

In the previous chapter we have shown that our solution domain reflects variability in the problem domain related to rule-based formalism and chaining strategy. Therefore, we need an atelier in which we can experiment with combinations of different rule-based languages — either based on logic or production rules, forward-chaining or backward-chaining — and different corresponding flavours of hybrid composition and hybrid aspects. In this section we present our atelier named *OReA*, which stands for *integrating Objects and Rules with Aspects*[1]. We start with presenting the architecture of OReA.

7.1.1 Architecture

A suitable basis for our atelier is Smalltalk, a dynamic, flexible and open development environment. It is a pure class-based object-oriented programming language, where everything is an object and all behaviour is accomplished with sending messages. Also, Smalltalk has built-in support for reflective programming.

Furthermore, our atelier should be outfitted with a number of initial elements that are easily extensible. Indeed, we want to keep development of state-of-the-art hybrid systems and aspect-oriented programming approaches to a minimum. This is desirable, not only for practical purposes, but also to ensure that the context from which we start is as realistic as possible. Therefore, we add two important components to our Smalltalk-based atelier: *SOUL*, a hybrid system which integrates Smalltalk with a Prolog-like logic programming language, and *AspectS*, an aspect-based approach for Smalltalk. A third important component which we use is reflection in Smalltalk.

[1] In modern Greek the interjection "orea" - $\omega\rho\alpha\iota\alpha$ - means *good* or *very well.*

A graphical overview of OReA's architecture is given in Fig. 7.1. This figure shows the three starting points of OReA: SOUL (top), AspectS (bottom left) and the reflective capabilities of Smalltalk (bottom right). We extend these components in OReA, discussed below.

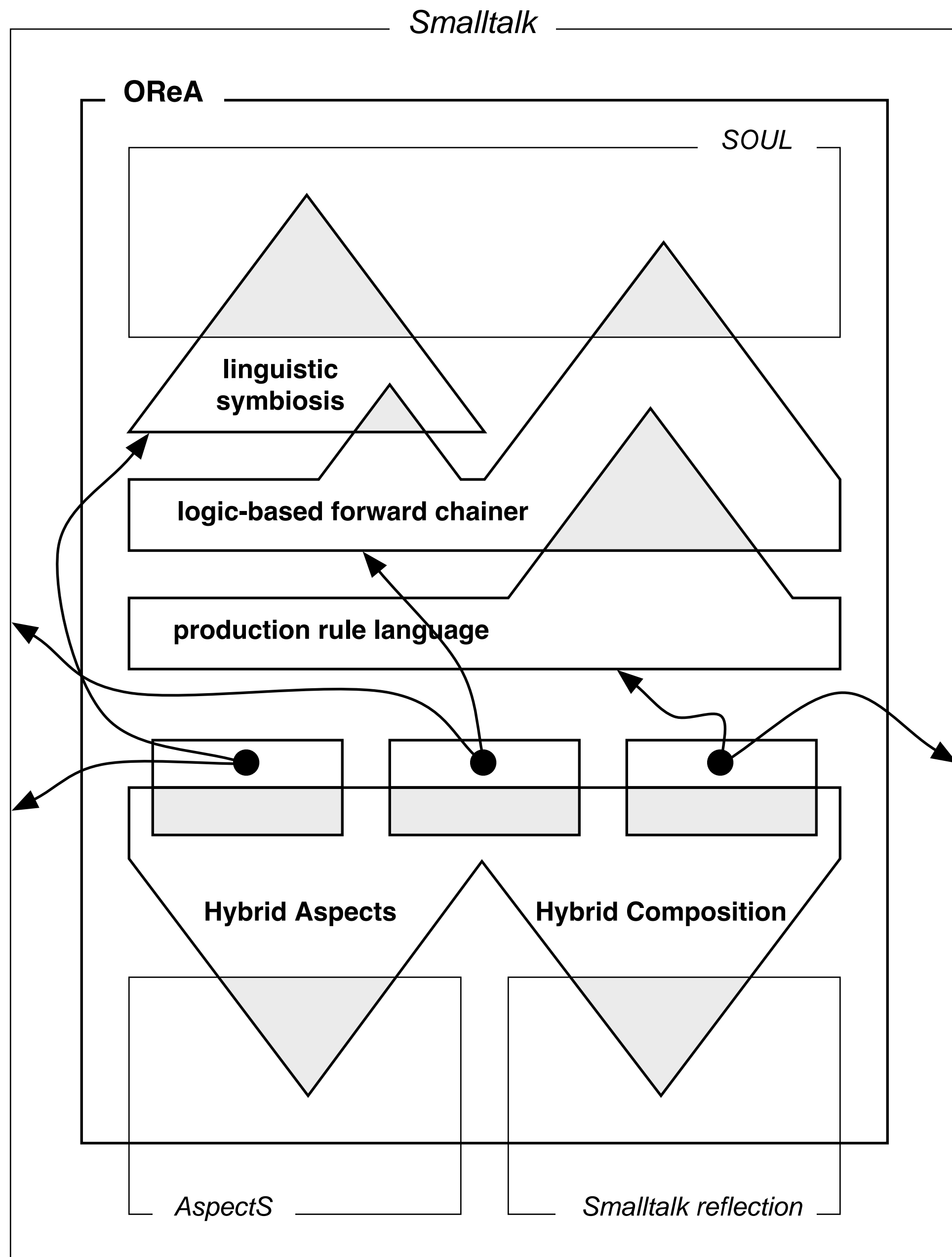


Figure 7.1: The architecture of OReA consisting of existing, basic components (Smalltalk, SOUL and AspectS) and our extensions (other rule-based languages, hybrid aspects and hybrid components).

First of all, let us consider the three rule-based languages in OReA. The first is SOUL's Prolog to which linguistic symbiosis with Smalltalk is added, depicted by the triangle which overlaps with SOUL. The second rule-based language, a logic-based forward chainer, reuses

parts of SOUL's original implementation and of the linguistic symbiosis we added. This is illustrated in the figure by the two triangular protrusions which overlap with SOUL and its linguistic symbiosis. The third rule-based language is based on production rules, also a forward chainer. The implementation of this language reuses many parts of the second rule-based language. Again, this is depicted by the triangular protrusion which overlaps the box of the logic-based forward chainer. The rule-based languages are presented in (section 7.1.2).

Next, we consider our hybrid composition and hybrid aspects in OReA, which use Smalltalk reflection and extend AspectS, respectively. AspectS is presented in (section 7.1.3). Each approach is depicted by a triangle which overlaps with the component they use or extend. That we combine hybrid composition and hybrid aspects is apparent in the figure because the two triangles are connected. There are three instances of our approach, which all have common ground but also a specific part. The figure illustrates this using the three boxes which overlap with the combination of hybrid composition and hybrid aspects. The first instance integrates Smalltalk and SOUL extended with linguistic symbiosis. This is depicted by the two arrows leaving the left-hand box and pointing to the Smalltalk box (outer box) and the triangle of SOUL's linguistic symbiosis. Likewise, the next two instances integrate Smalltalk and the logic-based forward chainer on the one hand, and Smalltalk and the production rule language on the other.

7.1.2 SOUL

The hybrid system *SOUL*, the *Smalltalk Open Unification Language*, is implemented in Smalltalk. SOUL is an implementation of a Prolog-like logic programming language in Smalltalk. It is originally developed by Roel Wuyts in the context of his doctoral dissertation to be a logic meta-programming language to support the co-evolution of design and implementation of object-oriented programs [Wuyts, 2001] [Wuyts, 1998]. But, over the years, a number of people have been involved in the development and use of SOUL as a research platform for applying logic programming to a number of software engineering problems: it has been used for explicitly representing domain knowledge in object-oriented applications [D'Hondt et al., 1999]; for checking, enforcing and searching for occurrences of programming patterns [Mens et al., 2001]; for supporting evolution of software applications [Mens & Tourwe, 2001]; and for architectural conformance checking [Wuyts & Mens, 1999].

SOUL is an excellent addition to our atelier. First of all, it has been around for a sufficiently long time to be stable and virtually bug-free. Moreover, its use in a wide range of applications by a number of different people has resulted in a well-designed system where each aspect of its functionality is explicitly and separately represented in order to promote reuse.

We have extended SOUL with a logic-based forward-chaining engine, reusing the representation of logic clauses and the unification algorithm. Furthermore, we have developed a new generation production rule language in SOUL. In both cases, only basic features are included and we opted for straightforward, if rather naive, implementations. The particularities of all three flavours of rule-based languages are presented below: SOUL's Prolog in section 7.1.2, our logic-based forward chainer in section 7.1.2 and our new generation production rule language in section 7.1.2.

One might argue that a logic-based forward-chaining engine in addition to Prolog's engine unbalances the rule-based languages in our atelier, since we do not have an backward-chaining equivalent of our production rule language. Yet, on the contrary, our atelier

enables us to investigate of effects of the chosen formalism and chaining strategy on the integration in isolated ways. Three different concretisations of our integration approach based on a combination of hybrid composition and hybrid aspects are developed: one for a language based on logic and backward-chaining, one for a language based on logic and forward-chaining, and one for a language based on production rules and forward-chaining. The first two concretisations allow a clear view on the differences imposed by the chaining strategy since the formalism is removed from the equation. Similarly, the second and third concretisations allow clarification on the differences imposed by the formalism since the chaining strategy is fixed. In addition to this, having a backward chainer and forward chainer of the same rule-based formalism leaves room for experimenting with how both chaining strategies can be integrated with object-oriented programming.

SOUL's Prolog

The survey presented in chapter 5, shows that SOUL, in its original incarnation, exhibits a form of language integration with Smalltalk which is similar to the other surveyed logic-based hybrid systems. Logic rules can use Smalltalk objects as values, but for sending messages to these objects the approach of embedding Smalltalk code snippets in the logic rules is used. The code snippets are delimited by `[` and `]`. They can contain rule variables which can be bound to Smalltalk objects. Smalltalk programs have to set up a SOUL rule engine explicitly via the API to activate rules. Queries are passed as strings or built using their representation as term objects in Smalltalk.

In OReA, we have adapted the syntax of logic clauses and predicates in SOUL in order to enable the linguistic symbiosis with Smalltalk. The actual linguistic symbiosis is explained in the next section on hybrid composition. Because the syntax for method identifiers and message selectors in Smalltalk is keyword-based, predicates now also use a keyword-based syntax. However, this does not change the semantics of the logic clauses in the least. For example, instead of using the more traditional logic-based syntax and writing

```
loyalCustomer(?customer) if hasChargeCard(?customer, ?card)
```

we write

```
?customer loyalCustomer if ?customer hasChargeCard: ?card
```

Although the logic sentences in this rule look like Smalltalk message sends, they are interpreted as predicates on logic variables as usual.

From the survey in chapter 5 we learn that in most systems language integration is facilitated by designing the rule-based language to be compatible with the object-oriented language. Therefore, we do not feel that adapting the logic-based language's syntax is too heavy a requirement for enabling the linguistic symbiosis needed by hybrid composition.

In order to illustrate the use of SOUL, suppose we want to infer all customers that are loyal. This is achieved by activating a query `?c loyalCustomer`, which results in the variable `?c` being bound to all inferred solutions. Each solution is a Smalltalk object, in this case representing a customer.

Extending SOUL with Logic-Based Forward Chaining

We extend SOUL with a simple logic-based forward chainer based on the algorithm presented in [Russel & Norvig, 1995] and shown in Fig. 7.2. Forward chaining is activated by asserting a fact f in the knowledge base KB. If f is already in the knowledge base, the

algorithm does nothing. If f is new, consider each rule that has a condition that unifies with f. For each such condition, if all the remaining conditions of the rule are in the knowledge base, then assert the substituted conclusion. The substitution θ keeps track of the way things unify.

```
procedure FORWARD-CHAIN(KB, f)

    if there is a fact in KB that is identical to f then return

    add f to KB

    for each (c1 ∧ ... ∧ cn ⇒ d) in KB such that for some i, UNIFY(ci, f) = ϑ succeeds do

        FIND-AND-INFER(KB, [c1, ..., ci-1, ci+1, ..., cn], d, ϑ)

    end
```

```
procedure FIND-AND-INFER(KB, conditions, conclusion, ϑ)

    if conditions = [ ] then

        FORWARD-CHAIN(KB, SUBST(ϑ, conclusion))

    else for each f' in KB such that UNIFY(f', SUBST(ϑ, FIRST(conditions))) = ϑ2 do

        FIND-AND-INFER(KB, REST(conditions), conclusion, COMPOSE(ϑ, ϑ2))
    end
```

Figure 7.2: Pseudo code for the forward-chaining algorithm of the logic-based forward chainer in OReA (adapted from [Russel & Norvig, 1995]).

For implementing this algorithm in SOUL, we reuse most of the internal representation of logic clauses (rules) of SOUL's Prolog. The new keyword-based syntax is also used. Furthermore, we were able to reuse the unification algorithm of the original SOUL. Nevertheless, forward-chaining rules are stored differently than backward-chaining ones, because they need to be indexed based on each condition rather than the conclusion.

In order to illustrate the way the forward chainer works, consider the aforementioned rule again, although this time, we use it as a forward-chaining rule.

```
?c loyalCustomer if ?c hasChargeCard: ?card
```

The forward chainer is activated by asserting a new fact. Suppose we assert that a customer *Jos* has a charge card *Visa*, represented by the fact `'jos' hasChargeCard: 'visa'`. The forward chainer then adds this fact to the rule base, which contains all rules and facts, and proceeds to determine all rules that are triggered by this new fact. The above rule is one of them, which means that it is instantiated with the values `'jos'` and `'visa'`. Then,

all triggered rules are fired one by one. Firing the above rule results in a new fact `'jos' loyalCustomer` being asserted. The forward chainer continues this process exhaustively until no more new facts have been asserted.

Note that the rule variables in the above example are bound to the strings `'jos'` and `'visa'`. As in the original SOUL, our forward-chaining rule variables can be bound to Smalltalk objects other than strings. This example is easier to write down with strings than with objects.

One could argue that our very straightforward implementation of the forward chainer would not result in a very performant or even adequate inference process. However, this textbook forward-chaining algorithm should be regarded as a black box which exhibits the standard "interface" and inference steps. Indeed it is activated with an assert and it has the same basic inference steps, establishing conditions and generating conclusions, as any other forward chainer. Other, more sophisticated forward chainers can therefore replace this one.

Extending SOUL with "New Generation" Production Rules

As is discussed in the previous chapter, we consider the next generation of production rule languages, similar to the ones found in existing hybrid production systems. In these production rule languages, rule variables are bound to objects and objects are stored in the system's working memory. Moreover, object-oriented features such as message sending are introduced in the production rule language in order to enable integration with the object-oriented language. We implement such a production rule language in OReA, reusing the implementation of our logic-based forward chainer.

The same rule which concludes the loyalty of a customer, is represented below in our production rule language:

```
if
  ?customer isa Customer.
  ?card isa ChargeCard.
  ?customer hasChargeCard: ?card.
do
  ?customer status: 'loyal'
```

When a new customer object *Jos* is asserted, it is added to the working memory and the system starts inferencing. Alternatively, the production system is activated when a customer object is already in the working memory and its state changes. Either way, all objects in the working memory are matched against the patterns (conditions) in the production rules. The above production rule is triggered when, aside from the customer object *Jos* (and possibly others), the working memory also contains at least one charge card object `Visa`, and both objects are matched successfully with the third pattern. Note that the third pattern is a *message send condition*, on of the object-oriented constructs with which the new generation production rule languages are outfitted. A similar construct is added to production rules' conclusions, resulting in *message send conclusion*. The message is sent for each substitution of matching working memory objects.

All triggered rules are stored in a conflict set. A conflict resolution strategy determines the rule which has to be fired. When the above rule is selected, the substituted conclusion is evaluated.

Note that we do not employ the RETE algorithm for optimising the very expensive iteration over all working memory objects and all production rules. Again, we argue that our production rule language is a black box. It is not important how each pattern is

instantiated, just that is done. With or without the RETE algorithm, the production system is still activated in the same way and exhibits the same basic inference steps.

7.1.3 AspectS

There also exist open implementations of aspect-oriented programming approaches in Smalltalk. *AspectS* [Hirschfeld, 2002] is an object-oriented framework for aspect-oriented programming: the object model of aspects and their constituents is reflected in the aspect definitions. It is an aspect-based approach: aspects consist of a pointcut expression and advice. AspectS employs a dynamic join point model. Weaving is dynamic since it is performed at run-time when an aspect is installed. Uninstalling an aspect results in it being “unweaved” dynamically. Weaving employs metaobject composition instead of code transformations, using *Method Wrappers*. A method wrapper replaces a compiled method (or another method wrapper) in the method dictionary of a class. When it is invoked it activates additional behaviour and possibly activates the wrapped method. AspectS coordinates the placement of these method wrappers according to the aspect definitions.

Originally, AspectS uses standard Smalltalk for expressing pointcuts, but a recent addition allows the use of SOUL for specifying pointcuts. Hence, SOUL is employed in two areas of our atelier: by AspectS (and our extensions to it) for defining pointcuts and as a basis for the concrete rule-based languages in OReA.

An aspect language typically has predefined *poincut designators* which each denote a specific kind of join point. There are several kinds of object-oriented join points considered by existing aspect-oriented programming approaches: method call, method reception and method execution. Typically there exists a pointcut designator for each of them. AspectS recognises one kind of object-oriented join point which coincides with the point right before a method is executed. As a result, AspectS has one predefined pointcut designator, which *targets* a method identifier *in* a class:

```
?jp targeting: ?selector in: ?class
```

This designator is actually a query which is evaluated in SOUL. The variables `?selector` and `?class` can be bound to actual values, whereas `?jp` has to remain unbound. The result of the query binds the inferred join points to the variable `?jp`. This result is then used by AspectS to install method wrappers on the specified join points.

For example, the query

```
?jp targeting: #addProduct: in: ShoppingCart
```

specifies all join points which target the execution of the method with identifier `#addProduct:` to instances of the class `ShoppingCart`.

We extend AspectS with additional pointcut designators to specify different kinds of rule-based join points. This is explained in section 7.3, when we introduce hybrid aspects in OReA.

Advice in AspectS is normal Smalltalk code which is executed before, after or around a join point.

Advice is parameterised with the interrupted execution context. This context is represented by an object in AspectS's framework. Thus, the context object can be accessed in the advice body to retrieve values from the interrupted execution.

7.2 Hybrid Composition in OReA

This section provides three concrete implementations of hybrid composition, each covering a variability with respect to rule-based formalisms and chaining strategies. Our three implementations of hybrid composition use the same object-oriented language, Smalltalk, but differ in the rule-based language being used. The latter are the rule-based languages provided by our atelier, OReA, presented in the previous section: the Prolog-like language of SOUL, a logic forward-chaining language and a new generation production rule language. The three corresponding implementations of hybrid composition are explained in section 7.2.1, section 7.2.2 and section 7.2.3. In each of these sections we provide simple examples which concretise hybrid composition. The ensuing section 7.2.4 discusses how hybrid composition is constrained to certain parts of the input programs.

7.2.1 Hybrid Composition of Smalltalk and Prolog

In chapter 6 we present the general correspondences between object-oriented and logic-based elements in hybrid composition. Moreover, we provide a scheme for matching logic-based and object-oriented elements using their qualifiers in order to achieve a linguistic symbiosis. In this section we concretise hybrid composition for integrating an object-oriented program in Smalltalk and a logic-based backward-chaining program in SOUL's Prolog. Our new keyword-based syntax for predicates is used, presented in section 7.1.2, which achieves the required one-to-one mapping with Smalltalk method qualifiers, which are also keyword-based.

Activating Backward-Chaining Logic Rules from Objects

At an object-oriented join point, hybrid composition activates corresponding logic-based behaviour whose qualifier matches the one of the object-oriented join point. Consider, for example, the Smalltalk method `addToShoppingCart:` defined in `Customer`, shown in code fragment 7.1. Adding a product to the shopping cart depends on whether the customer is allowed to buy the product. This is tested by sending the message `canBuy:` which results in a method execution.

```
Customer >> addToShoppingCart: p
  (self canBuy: p) ifTrue: [ shoppingCart addProduct: p ]
```

Code Fragment 7.1: The method `addToShoppingCart:` defined in the class `Customer`.

When this method with identifier `canBuy:` is going to be executed, a matching query with predicate `canBuy:` can be activated. The first argument of this query is the receiver of the method, a `Customer` object, and the second argument is the single parameter of the method, a `Product` object.

However, the question is, which one of the behaviours is going to be executed: the original method or the matching query, or both? In this implementation of hybrid composition, our strategy is to execute the original method if it is defined and activate the query if the method is not defined. This is possible because Smalltalk and SOUL check this at run time. If neither of the behaviours are implemented, a run-time error occurs.

Now suppose that our Smalltalk program does not implement a method named `canBuy:` in the class `Customer`. As a result, hybrid composition is going to activate the aforementioned query. This query can be proven, since there are a number of rules that infer whether

a customer can buy a product. These rules are shown in code fragment 7.2. They state that a customer can buy a product if he or she is located in Europe, or when this is not the case, if the product can be shipped internationally.

```
?c canBuy: ?p if ?c isLocatedInEurope.
?c canBuy: ?p if not(?c isLocatedInEurope) & ?p internationalShipping.
```

Code Fragment 7.2: Rules that conclude the predicate `canBuy:`.

When the object-oriented program activates a query, it expects a return value, especially since the query replaces the execution of a method. For example, as mentioned earlier, from the method `addToShoppingCart:` (code fragment 7.1) the query with predicate `canBuy:` is activated. Both query variables are bound, to a customer and a product object, respectively. Suppose the customer is located in Europe, which results in the first rule in code fragment 7.2 proving the query. This indication of success is transformed into the boolean value `true` and returned to the activation context as if it is the result of the message `canBuy:` being sent.

When the object-oriented program expects a value other than a boolean from an activated query, the unification operator `=` is used in rule conclusions. It unifies the left-hand side, a predicate, with the right-hand side, a value. Note that this approach is more compatible with the keyword-based syntax than adding an extra parameter, which also requires adding a new keyword. For example, consider the method in code fragment 7.3. It loops over all the purchased items in an order and adds their prices. Then, a discount percentage, obtained from the customer who places the order, is subtracted from the total price. Finally, the new total is returned.

```
Order >> calculatePrice
  | total |
  total := 0.
  self items do: [ :i | total := total + i price ].
  total := total - (total * customer discount / 100).
  ^total
```

Code Fragment 7.3: The method `calculatePrice:` in the class `Order`.

However, the method `discount` is not defined in `Customer`. The hybrid composition strategy first attempts to activate a query that matches with the undefined method `discount`, but no rules exist that conclude the predicate `discount`. An alternative query is automatically constructed and activated, one which uses the operator `=` to unify the `discount` predicate with an unbound variable: `<customer> discount = ?result`, where the first argument denotes the customer object. This query can be inferred by the rules in code fragment 7.4. Depending on the customer's type, i.e. premier, loyal or standard, a discount of 10, 5 or 0 percent is given, respectively. The customers' types are in turn inferred by other rules, not shown here.

Suppose the engine infers that the customer is loyal. In that case, the query is proven by the second rule and the query's `?result` variable is bound to the integer `5`. The interpretation of the operator `=` is to return the solution for this variable to the original object-oriented

```
?c discount = 10 if ?c premierCustomer
?c discount = 5 if ?c loyalCustomer
?c discount = 0 if ?c standardCustomer
```

Code Fragment 7.4: Rules that conclude the predicate `discount`.

execution context. Subsequently, the method in code fragment 7.3 subtracts the inferred discount percentage from the total price of the order.

A final issue in exchanging values between Smalltalk and SOUL is that logic rules do not make a distinction between having a single or multiple solutions, whereas methods have to be specific about returning a single object or a collection of objects. Because Smalltalk is dynamically typed, we choose the dynamic approached proposed in the previous chapter (section 6.6.2): when returning an inference result, whether single or multiple, it is wrapped in a *dual single item collection*, which supports the Smalltalk message protocol of collections. Alternatively, if the method expects a single result, the special collection is able to forward messages to the single (or first) result.

Sending Messages from Backward-Chaining Logic Rules

At a logic-based join point, hybrid composition activates corresponding object-oriented behaviour whose qualifier matches the one of the join point. Consider the rule in code fragment 7.5 which is needed by the rules in code fragment 7.2 for inferring their conditions. When inferring this rule's condition, the inference engine looks for at least one other rule which concludes `locationIs:`.

```
?c isLocatedInEurope if ?c locationIs: 'Europe'
```

Code Fragment 7.5: A rule that concludes `isLocatedInEurope`.

When the condition with predicate `locationIs:` is going to be inferred, a matching message send with selector `locationIs:` can be activated. The receiver of the message is the first argument of the condition's predicate, a Smalltalk object, and the single parameter of the message is the second argument of the predicate, the string `'Europe'`.

Again the question arises: which one of the behaviours is going to be executed: the original condition inference or the matching message send? We take the same approach as in activating queries from object-oriented programs. Our strategy is to continue with the original condition inference if a rule exists that concludes the condition's predicate, but send the matching message if no such rules exist.

When there are no rules that conclude the predicate `locationIs:`, hybrid composition sends the matching message to the object bound to the variable `?c`, a customer object [2]. The message send results in the method in code fragment 7.6 being executed.

```
Customer >> locationIs: aString
  ^shippingAddress country = aString
```

Code Fragment 7.6: The method `locationIs:` in the class `Customer`.

[2]Note that this variable has to be bound or this results in a run-time error.

The method in code fragment 7.6 returns either `true` or `false`. Since the result of inferring a condition is either success or failure, the result of the corresponding message send is converted to indicate the same.

In order to support more advanced interaction between Smalltalk and SOUL, more specifically to manipulate non-boolean message send results in SOUL, we introduce the unification operator `=` in conditions of rules. This approach is explained in our model in section 6.6.2. Again, the left hand side is a predicate and the right hand side what it should unify with. For example, consider the rule in code fragment 7.7, which is used, among others, by the discount rules shown earlier in code fragment 7.4. This is one of the rules for establishing the customer type, premier, loyal or standard. We can conclude that a customer is a premier customer if the total value of his or her purchases is greater than 10000. Obtaining this value is expressed in the first condition of the rule: the predicate `totalValueOfPurchases` about customers bound to the variable `?c` is unified with a variable `?p`. When a rule that concludes `totalValueOfPurchases` does not exist, the corresponding message is sent to the customer object(s) to which `?c` is bound.

```
?c premierCustomer if ?c totalValueOfPurchases = ?p & ?p > 10000
```

Code Fragment 7.7: A rule that concludes the predicate `premierCustomer`.

The message send results in the method `totalValueOfPurchases`, defined in the class `Customer`, being executed. This method calculates the sum of all the customer's past purchases. The result of this method is returned to the inference engine, which unifies it with the variable `?p`.

Our implementations of hybrid composition support the interpretation of a collection object, i.e. an object of a class in Smalltalk's `Collection` hierarchy, as multiple bindings for a rule variable. When returning an object of any other class to SOUL, it is interpreted as being a single binding of a rule variable.

Since Smalltalk is a dynamically typed language, we implement the dynamic approach to introducing unbound rule variables as explained in our model in section 6.6.2. Let us illustrate this feature with an example. Suppose products sold by the store are desktop computers, computers with integrated screens, laptop computers and monitors. These products have several characteristics such as brand and price for all hardware, space and speed for computers, size and resolution for screens in separate monitors and in all-in-one computers. A customer who wants to purchase a computer is able to search the available products using certain criteria, such as specific brands, price categories and so on. Additionally, it is possible to look for specific configuration such as multimedia computers or high-performance computers. When a customer is looking for a multimedia computer, the method in code fragment 7.8 is eventually invoked. This method sends the message `multimediaComputer` to the temporary variable `c`. The statement `|c|` is the standard way of declaring a temporary variable in Smalltalk. However, the next statement sends the message `multimediaComputer` to this temporary variable, which has not yet been initialised. This cause the variable being interpreted as an unbound rule variable. In this example, the query with predicate `multimediaComputer` and one unbound variable is activated.

This query is proven by the rules in code fragment 7.9. They state that something is a multimedia computer if it is a computer, it has a dvd player and it has a multimedia screen.

```
Store>>requestMultimediaComputer
  |c|
  c multimediaComputer.
  ^c
```

Code Fragment 7.8: The method `requestMultimediaComputer` in the class `Store`.

A computer has a multimedia screen if it is an all-in-one computer, has a screen and that screen is a multimedia screen, or if it is not an all-in-one computer, has a compatible screen and that screen is a multimedia screen. A screen is a multimedia screen if its size is greater than or equal to 17 inches.

```
?c multimediaComputer if
  ?c computer &
  ?c hasDVD &
  ?c hasMultimediaScreen.
?c hasMultimediaScreen if
  ?c allInOneComputer &
  ?c screen = ?s &
  ?s multimediaScreen.
?c hasMultimediaScreen if
  not(?c allInOneComputer) &
  ?c compatible: ?s &
  ?s multimediaScreen.
?s multimediaScreen if
  ?s size = ?x &
  ?x greaterOrEqual: 17.
```

Code Fragment 7.9: Rules related to concluding the predicate `multimediaComputer`.

The inferred computers are bound to the variable `?c` and the Smalltalk uninitialised temporary variable `c` *becomes* the collection of inferred computers. Note that the object-oriented program expects a collection as result and treats the temporary variable `c` as such. This is again enabled by inference results being *dual single item collections*, which can behave like a collection or delegate messages to the only (or first) object.

7.2.2 Hybrid Composition of Smalltalk and Logic Forward Chaining

This section presents hybrid composition with Smalltalk and the forward chainer with which SOUL is extended. The linguistic symbiosis is similar to the one employed by the previous flavour of hybrid composition, since both are based on the logic formalism.

Activating Forward-Chaining Logic Rules from Objects

Earlier, when discussing hybrid composition with the logic-based backward chainer, we explain how it activates a corresponding query from an object-oriented program. In a logic-based forward chainer, the same strategy and matching is applied, but the corresponding behaviour being activated is an assert. For example, let us consider again the method `addToShoppingCart:` in the class `Customer`, shown in code fragment 7.1. If a customer is

allowed to buy a product, the method delegates the actual addition of the product to his or her ShoppingCart object. At least, that is what it looks like in this method's definition. However, suppose there is no method addProduct: defined in the class ShoppingCart. In that case, hybrid composition determines the corresponding assert to be activated, since the integration is with a logic-based forward chainer this time. An assert is activated with the predicate addProduct:, the receiver of the method as the first argument, a ShoppingCart object, and the single parameter of the method as the second argument, a Product object.

```
?cart addProduct: ?best  if
  ?cart addProduct: ?product &
  ?product isBook &
  ?best bestsellingBook
?cart addProduct: ?batteries  if
  ?cart addProduct: ?product &
  ?batteries batteriesFor: ?product
?cart addProduct: ?wrapper if
  ?cart addProduct: ?product &
  ?wrapper giftWrapperFor: ?product
```

Code Fragment 7.10: Rules that conclude the predicate addProduct.

The forward-chaining rules in code fragment 7.10 are activated by the assert since they have conditions which possibly unify with the newly asserted fact. The first rule expresses that for all books that are added to a shopping cart, all bestsellers — but typically this is only one — are also added to the cart [3]. The second rule states that for all products added to a shopping cart, batteries for each product are also added. Some products, such as books, wraps and batteries themselves, have no batteries. Hence, the condition with predicate batteriesFor: will not infer batteries for these products. The third rule expresses that for all products added to a shopping cart, a gift wrapper for each product is also added to the cart. Again, inferring the condition with predicate giftWrapperFor: will not be successful for products which do not have a wrapper, such as batteries and wrappers.

Note that the forward chainer indeed does not "return" any explicit inference results. The new facts are generated as a result of concluding a rule and are added to the knowledge base.

Sending Messages from Forward-Chaining Logic Rules

In a forward chainer, there are two kinds of join points that may result in a message send: condition inference and conclusion generation. The first one is similar to the join point in logic-based backward chainers. Hence we refer to section 7.2.1 for an illustration. The second kind of join point, a conclusion generation, also corresponds to a message send. Apart from the logic-based join point being a different one, the matching is achieved in the same way as before. For example, consider the rule in code fragment 7.11 which recommends products to a certain customer. It infers the products to be recommended in the following way: whenever a product is added to a shopping cart, infer all the other customer who purchased this product earlier and infer all the other products these other customers once purchased. The second condition simply retrieves the customer from the cart for whom

[3] Note that when adding multiple books, the fact that a bestseller is also added will be inferred only once since our forward-chaining logic-based language does not conclude facts that already exist.

the recommendations are inferred. The rule concludes a relationship between this customer and the inferred recommendations with the predicate `recommend:`.

```
?customer recommend: ?otherProduct if
  ?cart addProd: ?product &
  ?cart customer = ?customer &
  ?product purchasedBy: ?otherCustomer &
  ?otherCustomer purchased: ?otherProduct
```

Code Fragment 7.11: A rule that concludes the predicate `recommend:`.

Conclusions with predicate `recommend:` correspond to the matching message send with selector `recommend:`. However, we need to establish a hybrid composition strategy that decides when the original conclusion is generated and when the matching message is sent. The strategy that is implemented here is to attempt first to send the message. If there is no method with that signature defined in the class of the receiver of the message, the conclusion is generated instead.

Suppose that in our example, a method named `recommend:` is defined in the class of the objects to which the rule variable `?customer` is bound, as shown in code fragment 7.12. In this case, hybrid composition sends a message with selector `recommend:` for each set of bindings. The `recommend:` method delegates to the instance variable of the customer, `recommendations`, the addition of the recommended product to the existing recommendations. This is achieved by ranking the recommended products according to the number of times they have been recommended.

```
Customer >> recommend: aProduct
  recommendations add: aProduct
```

Code Fragment 7.12: The method `recommend:` in the class `Customer`.

The previous issue establishes that there are two kinds of logic-based join points that can result in a message being sent. Nevertheless, only the first kind, inferring a condition, actually expects a result. The second kind of join point, generating a conclusion, does not expect a result, either implicit or explicit, of the behaviour that is activated, either actually generating the conclusion or sending messages instead.

The way message send results are returned to condition inferences in this manifestation of hybrid composition, is exactly the same as in hybrid composition of Smalltalk and SOUL's Prolog, presented in section 7.2.1.

7.2.3 Hybrid Composition of Smalltalk and Production Rules

This section presents the implementation of the last flavour of hybrid composition.

Activating Forward-Chaining Production Rules from Objects

Since production systems are data-oriented, they are activated when new data is asserted. When production rules are integrated with an object-oriented program, the creation of new objects results in the objects being asserted. Note that hybrid composition is constrained to certain classes, which also limits the instances that are actually asserted in the working

memory. We discuss this feature of hybrid composition later on in section 7.2.4. Once objects are in the working memory, they are monitored for state changes. When an instance variable changes or an element is added or removed in a collection instance variable, the rules are activated again.

To illustrate this, consider the example method in code fragment 7.13. This method named `createCustomerNamed:withChargeCard:` is defined in the class `Store`. It is responsible for creating a customer object and setting its name and charge card. The instance creation is achieved by sending the message `new` to the class `Customer`. The name, a string, is the first parameter of this method. The charge card, an instance of the class `ChargeCard`, is the second parameter of the method. Finally, the store delegates to itself the task of storing the customer, which might involve adding it to a collection of available customers and making the object persistent.

```
Store >> createCustomerNamed: name withChargeCard: card
  | customer |
  customer := Customer new.
  customer name: name.
  customer chargeCard: card.
  self storeCustomer: customer.
  ^customer
```

Code Fragment 7.13: The method `createCustomerNamed:withChargeCard:` in the class `Store`.

The creation of an instance of `Customer` results in this object being asserted. However, since the newly created object's state is not yet initialised, activating the rules at this point is not very useful. Instead, an an optimisation, our implementation of hybrid composition merely adds the object to the working memory, but does not activate the rules as a normal assert would. Since the object is initialised after its creation, such as with the mutators `name:` and `chargeCard:` in the example, this activates the rules since the production system monitors the object.

Sending Messages from Forward-Chaining Production Rules

As mentioned earlier, modern-day hybrid production systems incorporate object-oriented features such as sending messages to production rule variables which are bound to objects. We again stress that this is common practice and in no way a contribution of ours.

Because our production rule language is a forward chainer, two kinds of rule-based join points are again considered: condition inference and conclusion generation. This is analogous to the logic-based forward chainer presented in section 7.2.2.

As an example of sending messages in production rules, consider the production rule in code fragment 7.14. This rule is activated, among others, when a customer's instance variables are changed, as does the method `createCustomerNamed:withCharge-Card:` in code fragment 7.13. The first two condition statements match all objects in the working memory of the classes `Customer` and `ChargeCard`, respectively. The third condition, on the other hand, is a message send condition and an example of the first kind of join point. It can be mapped unambiguously to a message with selector `hasChargeCard:`, sent to all objects bound to the rule variable `?customer` with all corresponding objects bound to `?card` as parameter. If the receivers of this message are instances of the class `Customer`, the method in code fragment 7.15 is executed.

```
if
  ?customer isa Customer.
  ?card isa ChargeCard.
  ?customer hasChargeCard: ?card.
do
  ?customer status: 'loyal'
```

Code Fragment 7.14: A production rule that concludes the status of a customer.

```
Customer >> hasChargeCard: card
  chargeCard = card
```

Code Fragment 7.15: The method `hasChargeCard:` in the class `Customer`.

The production rule in code fragment 7.14 also exemplifies the second kind of production rule join point, the one occurring in the conclusion of a production rule. In this case, the conclusion consists of a conclusion message send statement with one rule variable `?customer` and selector `status:`. Similar to condition message send statements, this results in the matching message send being evaluated in the object-oriented language.

The way message send results are returned to condition inferences in this implementation of hybrid composition is similar to the way the other implementations of hybrid composition take care of this issue. As is illustrated by the production rule in code fragment 7.14, condition inferences expect an indication of success or failure. Therefore, the messages being sent should return Smalltalk boolean values `true` or `false`. Alternatively, when a non-boolean result is expected from a message send, the operator `=` is used in a condition to bind the rule variable on the right-hand side to the message send results. Again, objects of a class in Smalltalk's `Collection` hierarchy are interpreted as multiple bindings for the variable. The use of this operator is illustrated in code fragment 7.16. The second condition sends the message `items` to each shopping cart object to which the variable `?cart` is bound. This accessor returns a collection of products, which is interpreted as multiple matches for the `?product` variable. The third condition sends the message `giftWrapper` to each of the products to which `?product` is bound. This accessor returns a wrapper object, to which the variable `?wrapper` is bound. If the receiver does not understand the message `giftWrapper`, the condition fails to be established.

```
if
  ?cart isa ShoppingCart.
  ?cart items = ?product.
  ?product giftWrapper = ?wrapper.
do
  ?cart add: ?wrapper
```

Code Fragment 7.16: A production rule that concludes gift wrappers to be added to a shopping cart.

7.2.4 Constraining Hybrid Composition

One of the advantages of hybrid composition is that it allows the developer to specify the parts of the input programs that are considered for composition. The syntax for expressing this is given below:

$composition \rightarrow compositionRule\ \{packageName\}^{+}$ **and** $\{ruleLayerName\}^{+}$
$compositionRule \rightarrow$ **composeByName** | **composeWithReturnValue** | **...**

For example, the statement below signals hybrid composition that the object-oriented package with name `ECom` is to be considered for composition, as well as the rule layers with names `DiscountRules` and `RefundRules`.

```
composeByName ECom and DiscountRules RefundRules
```

Hybrid composition is able to figure out what kind of rules are in those rule layers, in this case logic-based backward-chaining rules, because this is indicated when the rules are defined. Then, the correct flavour of composition is initiated. There is no fundamental reason why hybrid composition should not be able to compose different kinds of rules, provided that ambiguities in the activation of rules are resolved in the rules themselves or by the hybrid composition. Our implementation, however, does not allow different kinds of rules to be composed.

7.3 Hybrid Aspects in OReA

Analogous to the previous section on hybrid composition, this section presents hybrid aspects for Smalltalk and the same three rule-based languages: SOUL's Prolog, a logic-based forward chainer and a new generation production system. We reuse the examples from the previous section on hybrid composition as much as possible so that the reader may compare the approaches. We do not yet show how the two approaches collaborate, but address this in the next chapter where we evaluate our approach.

Note that all the abstract grammar elements introduced in the previous chapter in section 6.7 are covered in the following, albeit in a less structured way. Typically, the following sections discuss query, message send and unification/match statements explicitly, whereas the other elements of our hybrid aspect languages come up implicitly in these sections.

7.3.1 Hybrid Aspects for Smalltalk and a Prolog-like Language

Rule-Based Pointcut Designators

We extend AspectS with a predefined, rule-based pointcut designator for specifying the inferences of conditions, similar to its predefined, object-oriented pointcut designator presented in section 7.1. The rule-based pointcut designator is represented as a query in SOUL, similar to the object-oriented one.

```
?jp inferring: ?predicate
```

where all but the `?predicate` variable is bound to actual values. The result of this query binds the variable `?jp` to the inferred join points. This result is then used for weaving hybrid advice at the specified join points. For example,

```
?jp inferring: #bestsellingBook
```

specifies all join points which denote the inference of a condition with predicate `bestsellingBook`.

Advice Argument

The context of each join point, object-oriented or rule-based, is exposed to the hybrid advice. In OReA, the context of an object-oriented join point consists of message selector, receiver and arguments. The context of a rule-based join point consists of a predicate, arguments and a set of bindings. We have defined a set of polymorphic accessors for both kinds of contexts, shown in table 7.1. The columns show for each accessor what the rule-based and object-oriented contexts return. We have chosen for a very simple and uniform interface to the different kinds of execution contexts. More meaningful accessor names are feasible but might be confusing due to the discrepancy between method parameters and predicate arguments: the first argument of a predicate corresponds to the receiver of a method, the second argument corresponds to the first parameter, etc.

Table 7.1: Uniform interface of rule-based and object-oriented contexts.

	`at: 1`	`at: 2`	...	`at:` n
rule-based context	1st arg	2nd arg	...	n^{th} arg
object-oriented context	receiver	1st par	...	$n-1^{th}$ par

For example, when the logic-based program execution is interrupted at a join point denoted by the following pointcut

```
?jp inferring: #totalValueOfPurchases:
```

the hybrid advice can access the context of the interrupted execution. The accessor `at:` with parameter `1` returns the first argument, whereas the accessor `at:` with parameter `2` returns the second argument.

Similarly, when the object-oriented program execution is interrupted at a join point denoted by the following pointcut

```
?jp targeting: #calculatePrice in: Order
```

the accessor `at:` with parameter `1` returns the receiver of the interrupted method execution, an instance of `Order`.

Query

Query expressions in hybrid advice are preceded by a special character, a question mark. Consider, for example, the method `addToShoppingCart:` defined on the class `Customer` with one parameter `product` in code fragment 7.17. It simply delegates the actual addition of a product to the shopping cart, an attribute of customer.

```
Customer>>addToShoppingCart: product
  shoppingCart addProduct: product
```

Code Fragment 7.17: The method `addToShoppingCart:` defined in the class `Customer`.

Now consider the rules for inferring if a customer can buy a certain product in code fragment 7.18. They state that a customer can buy a product if he or she is located in Europe, or when this is not the case, if the product can be shipped internationally.

```
?c canBuy: ?p if ?c isLocatedInEurope.
?c canBuy: ?p if not(?c isLocatedInEurope) & ?p internationalShipping.
```

Code Fragment 7.18: Rules that conclude the predicate `canBuy:`.

In order to integrate these rules with the method, we write the hybrid aspect in code fragment 7.19. We review the lines below, which are numbered. Note that in this first example of an hybrid aspect, we did not nest the expressions for the sake of clarity. Later examples of hybrid aspects will be more succinct.

```
1. ?jp targeting: #addToShoppingCart: in: Customer
2. :context
3.   | customer product |
4.   customer := context at: 1.
5.   product := context at: 2.
6.   customer ?canBuy: product.
7.   _proceed
```

Code Fragment 7.19: The hybrid aspect that integrates the method `addToShoppingCart:` defined in `Customer` and rules that conclude the predicate `canBuy:`.

line 1 The first line contains an object-oriented pointcut using the pointcut designator `targeting:in:` to denote the execution of the method named `#addToShoppingCart:` in the class `Customer`.

line 2 The second line declares the name of the hybrid advice argument, which is bound to the object representing the interrupted execution context. Note that the colon preceding the argument name `context` originates from *block* parameters in Smalltalk, because in AspectS advice is implemented as such.

line 3 The third line consists of temporary variable declarations, `customer` and `product`.

The next two lines (**Line 4** and **Line 5**) are statements that initialise the temporary variables. This is done in a standard object-oriented manner, using an assignment. The right-hand side of both assignments access the interrupted execution context using the accessors presented earlier in table 7.1.

line 4 This line illustrates what we explained earlier: the receiver is retrieved from the interrupted context with the accessor `at:` and parameter `1`. The result is a customer object, which is assigned to the temporary variable `customer`.

line 5 This line is similar to the previous line: it retrieves the second parameter of the context, a product object, with the accessor `at:` and assigns it to the temporary variable `product`.

line 6 This line shows the activation of the query with predicate `canBuy:`. The predicate name is preceded with `?`. The arguments of this query are the customer and the product, respectively. After the query has been performed, success or failure is indicated which results in the next statement being evaluated or not.

line 7 This line illustrates the proceed expression `_proceed` which allows the interrupted method execution to proceed. In this example, if the query is successful, the interrupted execution of the method `addToShoppingCart:` is allowed to proceed.

The question mark distinguishes query statements from message send statements. The reason for clearly marking the different behaviour activation statements is that we require complete control over what is exactly activated in hybrid advice. It must be unambiguously clear that a query is activated or a message is sent.

Rule variables can be bound by using them in queries. In order to illustrate this, consider the method `calculateSubTotal` defined on the class `Order` in code fragment 7.20. It iterates over all the purchased items in an order and adds their prices.

```
Order >> calculatePrice
  | total |
  total := 0.
  self items do: [ :i | total := total + i price].
  ^total
```

Code Fragment 7.20: The method `calculatePrice:` in the class `Order`.

Now consider the rules for inferring a customer's discount in code fragment 7.21. Depending on the customer's type, i.e. premier, loyal or standard, a discount of 10, 5 or 0 percent is given respectively. The customers' types are in turn inferred by other rules, not shown here.

```
?c discount: 10 if ?c premierCustomer.
?c discount: 5 if ?c loyalCustomer.
?c discount: 0 if ?c standardCustomer.
```

Code Fragment 7.21: Rules that conclude the predicate `discount`.

In order to integrate these rules with the Smalltalk program that calculates the price of an order, we write the hybrid aspect in code fragment 7.22. We review the lines below.

```
1. ?jp targeting: #calculatePrice in: Order
2. :c
3.   | total |
4.   total := _proceed.
5.   (c at: 1) customer ?discount: percentage.
6.   total - (total * percentage / 100)
```

Code Fragment 7.22: The hybrid aspect that integrates the method `calculatePrice` defined in `Order` and rules that conclude the predicate `discount:`.

lines 1 to 3 These lines are again similar to the ones shown in the previous hybrid aspects.

lines 4 This line shows another use of the _proceed statement. This time its result, which is the result of executing the method calculatePrice in an instance of Order, is assigned to the temporary variable total.

line 5 This line shows the activation of a query with predicate discount:. The first argument of this query is the customer which is accessed from the order object, which is in turn retrieved from the context. The second argument of this query is the uninitialised temporary variable percentage. When used in a query, it is considered to be an unbound rule variable. After the query has been performed, this variable is bound to the inferred discount percentage, more specifically 0, 5 or 10. Since the rule variable is bound to one object, the evaluation of the next statements does not branch.

line 6 When the temporary variable percentage is a parameter in the message send expression in this line, its binding is looked up and used. The new total price is calculated by subtracting the percentage from the original total price.

Message Send

Suppose that the rules that conclude canBuy: in code fragment 7.18 need to be integrated with a method for inferring the condition with predicate isLocatedInEurope. The method that could be used to achieve this, isLocatedIn: defined in Customer, is show in code fragment 7.23. It has one parameter region which is a string denoting the name of the region. The method returns the result of comparing this string with the customer's location, stored in the attribute location.

```
Customer>>isLocatedIn: region
  ^location = region
```

Code Fragment 7.23: The method isLocatedIn: defined in the class Customer.

The hybrid aspect in code fragment 7.24 illustrates how the integration is achieved. Again, we review the lines of code below.

```
1. ?jp inferring: #isLocatedInEurope
2. :c
3.   | customer |
4.   customer := c at: 1.
5.   customer isLocatedIn: 'Europe'
```

Code Fragment 7.24: The hybrid aspect that integrates rules that conclude isLocatedInEurope: and the method isLocatedIn: defined in the class Customer.

line 1 The first line contains a rule-based pointcut using the pointcut designator inferring: to denote the inference of conditions with predicate #isLocatedIn-Europe.

line 2 The second line again declares the name of the context, c this time.

line 3 The third line declares the only temporary variable, `customer`.

line 4 This line retrieves the necessary value from the interrupted execution context `c`, the first argument of the predicate `isLocatedInEurope`. This value is a rule variable which is bound to a customer object.

line 5 The last line of this hybrid aspect contains the actual message send expression. The message with selector `isLocatedIn:` is sent to the rule variable that was retrieved in the previous line from the rule-based execution context. Since the rule variable is used in a message send, its binding, a customer object, is retrieved first. The hybrid aspect in returns the result of evaluating the last expression to the interrupted execution context, which interprets it as an indication of success or failure in inferring the condition. Note that this hybrid aspect replaces the original execution.

Unification

Unbound rule variables are introduced in hybrid advice when they are retrieved from the context with the accessors of table 7.1. Typically, when hybrid advice replaces a condition inference containing an unbound variable, it is expected to bind it. One way to achieve this is using the unification expression with operator `?=`. We illustrate this using the example rule in code fragment 7.25. It is clear that when the third condition with predicate `compatible:` is going to be inferred, the variable `?s` is unbound.

```
?c hasMultimediaScreen if
  not(?c allInOneComputer) &
  ?c compatible: ?s &
  ?s multimediaScreen.
```

Code Fragment 7.25: A rule that concludes the predicate `hasMultimediaScreen`.

Consider the method presented in code fragment 7.26. This method basically implements a request to the store for all screens compatible with the receiver, a computer. It returns the result, a collection of screen objects.

```
Computer>>getCompatibleScreens
  ^store compatibleScreensFor: self.
```

Code Fragment 7.26: The method `getCompatibleScreens` in the class `Computer`.

The hybrid aspect in code fragment 7.27 takes the following steps:

lines 1 and 2 are similar to the ones shown in the previous hybrid aspects.

line 3 This line contains the message send expression with selector `getCompatibleScreens` and receiver the binding of the rule variable retrieved from the context, a computer object. The result of this message send, a collection object, is unified with the unbound rule variable which is the second argument of the interrupted condition inference. The hybrid advice returns the result of this unification, success or failure, which is interpreted as the result of inferring the condition. A side effect of this hybrid aspect is that the variable `?s` in the rule in code fragment 7.25 is bound to the collection of screens obtained by the method `getCompatibleScreens`. Note that this collection is interpreted as multiple solutions for the variable `?s`.

```
1. ?jp inferring: #compatible:
2. :context
3.   (context at: 2) ?= (context at: 1) getCompatibleScreens
```

Code Fragment 7.27: The hybrid aspect that integrates rules that conclude `hasMultimediaScreen` and the method `getCompatibleScreens:` defined in `Computer`.

7.3.2 Hybrid Aspects for Smalltalk and a Forward-Chaining Logic Language

A forward chainer requires another kind of pointcut designator and hybrid aspects typically activate asserts instead of queries. Note that the same reflective access to the execution context is provided as in hybrid aspect for SOUL's Prolog.

Rule-Based Pointcut Designators

The second kind of rule-based join point, only used by forward chainers, occurs when a conclusion is being generated. As we did for the condition inference join point presented in section 7.3.1, we provide a predefined pointcut designator for specifying conclusion generation join points. Again, it is represented as a query:

```
?jp concluding: ?predicate
```

For example,

```
?jp concluding: #inCart:
```

specifies all join points which denote conclusions with predicate `#inCart:`.

Assert

Hybrid advice for integrating Smalltalk and a logic forward chainer has exactly the same ingredients as hybrid advice in the context of the logic backward chainer, presented in section 7.3.1. The only difference is that rule-based behaviour is activated with an assert instead of with a query. Analogously to the query activation statements, the predicate of the assert is preceded with a special character, an exclamation mark this time, in order to distinguish it from message send statements.

The next example illustrates how hybrid advice activates the logic forward chainer. Consider the method named `addProduct:` in code fragment 7.28, defined in class `ShoppingCart`, which adds a product to a shopping cart's collection of items.

```
ShoppingCart >> addProduct: aProduct
  ^items add: aProduct
```

Code Fragment 7.28: The method `addProduct:` in `ShoppingCart`.

The other input program consists of forward-chaining rules that infer products that should be in a shopping cart based on the contents of the cart. These rules are shown in code fragment 7.29.

The first hybrid aspect is given in code fragment 7.30.

```
?best inCart: ?cart  if
  ?product inCart: ?cart &
  ?product isBook &
  ?best bestsellingBook
?batteries inCart: ?cart  if
  ?product inCart: ?cart &
  ?batteries batteriesFor: ?product
?wrapper inCart: ?cart if
  ?product inCart: ?cart &
  ?wrapper giftWrapperFor: ?product
```

Code Fragment 7.29: Rules that conclude the predicate `inCart:`.

```
1. ?jp targeting: #addProduct: in: ShoppingCart
2. :context
3.   _proceed !inCart: (context at: 1)
```

Code Fragment 7.30: The hybrid aspect for integrating the method `addProduct:` in `ShoppingCart` with rules that conclude the predicate `inCart:`.

line 3 This statement achieves many things. Foremost it activates the rules in code fragment 7.29 by the assert of the predicate `!inCart:`. As with all behaviour activating statements, the arguments are evaluated before the behaviour itself is activated. First, the aforementioned `_proceed` statement is evaluated which allows the original, interrupted execution to proceed. This results in the execution of the `addProduct:` method in code fragment 7.28. It returns the product that it just added to the shopping cart's collection of items. This product object is used as first argument in the assert. Secondly, the context accessor is evaluated which returns the shopping cart object.

Message Send

In order to demonstrate the use and effect of the second kind of rule-based pointcut designators, we continue with the same example. The hybrid aspect shown in code fragment 7.31 integrates the two input programs in code fragment 7.28 and 7.29 in the other direction.

```
1. ?jp concluding: #inCart:
2. :context
3.   (context at: 2) addProduct: (context at: 1)
```

Code Fragment 7.31: The hybrid aspect for integrating rules that conclude the predicate `inCart:` with the method `addProduct:` defined in `ShoppingCart`.

line 3 At each conclusion generation join point denoted by the pointcut in **line 1**, the hybrid advice sends a message `addProduct:` to the second argument of the interrupted conclusion predicate, which is a rule variable bound to a product object, with the first argument as parameter, which is a rule variable bound to a shopping cart object.

Note that the bindings are passed to the message send, and not the rule variables themselves.

The two hybrid aspects ensure an interesting interaction between the two input programs. Running the integrated input programs with a customer who adds a book and a walkman to his shopping cart, results in the following products being subsequently added: the book, a book gift wrapper for the book, the bestseller (there is only one), a book gift wrapper for the bestseller, the walkman, a standard battery pack for the walkman and a small electronics gift wrapper for the walkman. Recall that wrappers for wrappers and batteries do not exist, and neither do batteries for books and batteries. Therefore, the rules do not loop.

Unification

Consider the method `createOrder:` defined in the class `Store`, shown in code fragment 7.32. In short, this method creates an order for a customer, which consists of setting all sorts of information in the order, taken from the customer and the customer's shopping cart. It calculates the price of the order (using the method in code fragment 7.20), stores the order, empties the shopping cart and finally returns the order.

```
Store >> createOrder: c
  | o |
  o := Order new customer: c;
    orderDate: Date today;
    items: c shoppingCart items;
    shippingAddress: c shippingAddress;
    billingAddress: c billingAddress;
    calculatePrice.
  self storesOrder: o.
  c shoppingCart empty.
  ^o
```

Code Fragment 7.32: The method `createOrder:` defined in the class `Store`.

Now consider the hybrid aspect in code fragment 7.33. This integration reuses the rules that conclude `inCart:` given in code fragment 7.29. However, instead of integrating them with the `addProduct:` method in code fragment 7.28 as is achieved with the hybrid aspect in code fragment 7.30, we integrate them instead with the method `createOrder:` in code fragment 7.32.

line 5 This line retrieves the items from the customer's shopping cart. The unification operator `?=` unifies the resulting object with the temporary variable `product`, which is an unbound rule variable because it is uninitialised. Because the items in a shopping cart are stored in a collection object, the unification operator interprets it as multiple solutions for the unbound variable `product`. This results in the evaluation of the next statements to be branched for each set of bindings.

line 6 In this line an assert with predicate `inCart:` is activated. This statement is evaluated with each product object to which `product` is bound.

```
1. ?jp targeting: #createOrder: in: Store
2. :c
3.   | cart product |
4.   cart := (c at: 2) shoppingCart.
5.   product ?= cart items.
6.   product !inCart: cart.
7.   _proceed.
```

Code Fragment 7.33: The hybrid aspect that integrates the method `createOrder:` defined in `Store` with rules that conclude the predicate `inCart:`.

7.3.3 Hybrid Aspects for Smalltalk and a Forward-Chaining Production Rule Language

The last flavour of hybrid aspects considers Smalltalk and a production rule language extended with object-oriented message send statements. The same kinds of pointcut designators and language integration issues need to be considered as in the previous flavour of hybrid aspects, since both take forward chainers into account. However, variety is introduced by the difference in rule-based formalism. Additionally, hybrid advice has to be able to delete objects from a production system's working memory.

Rule-Based Pointcut Designators

Since the production rule language is a forward chainer, we need two kinds of rule-based pointcut designators: one for inferring a condition and one for generating a conclusion. They target condition and conclusion message sends, respectively:

```
?jp inferring: ?selector in: ?class
```

and

```
?jp concluding: ?selector in: ?class
```

Assert

Activating the production system is achieved by asserting an object. This requires hybrid advice that is very similar to the one for logic forward chainers. However, instead of having an assert statement that is qualified with a predicate, the assert statement for production rule languages consist of an expression which evaluates to the object being asserted. This time, the exclamation mark is not used for distinguishing assert statements from message send statement, but as an assert *operator*.

We want to integrate the `createOrder:` method previously shown in code fragment 7.32 with production rules like the one in code fragment 7.16 for inferring other products to be added to the shopping cart, before the order is created. To achieve this, we define the hybrid aspect in code fragment 7.34.

line 3 This line illustrates asserting an object with the `!` operator. First of all, the expression ensuing the operator is evaluated. The method's parameter, a customer, is retrieved from the context. The accessor `shoppingCart` is sent to the customer, which retrieves his or her shopping cart. This object is the argument of the assert operator `!`.

```
1. ?jp targeting: #createOrder: in: Store
2. :c
3.   !(c at: 2) shoppingCart.
4.   _proceed
```

Code Fragment 7.34: The hybrid aspect for integrating the method `createOrder:` defined in `Store` with production rules that conclude products to be added to a shopping cart.

Message Send

The second hybrid aspect integrates alternative object-oriented code with the production rule in code fragment 7.16. It is shown in code fragment 7.35.

```
1. ?jp inferring: #giftWrapper in: Product
2. :context
3.   XmasWrapper for: (context at: 1)
```

Code Fragment 7.35: The hybrid aspect for integrating an alternative wrapper with a production rule that adds gift wrappers for products to a shopping cart.

line 3 This line returns a new `XmasWrapper` for the product in the interrupted production rule context to the interrupted production rule execution. As such, it overrides the conclusion message send `giftWrapper` that retrieves a normal gift wrapper for a product.

One of the operations provided by a production system is deleting objects from the working memory. This operation has no effect other than the particular object being deleted. Hybrid advice for weaving object-oriented programs and production rules thus foresees an operator, more specifically the back quote ‘, that deletes the object to which the preceding expression evaluates. For example, the previous hybrid aspect in code fragment 7.34 is extended, shown in code fragment 7.36, with a delete statement in **line 6**.

```
1. ?jp targeting: #createOrder: in: Store
2. :context
3.   | cart |
4.   cart := (context at: 2) shoppingCart
5.   ! cart.
6.   ‘cart.
7.   _proceed
```

Code Fragment 7.36: The hybrid aspect with delete operator for integrating the method `createOrder:` defined in `Store` with production rules that conclude products to be added to a shopping cart.

Match

Although the interrupted execution context of production rules does not contain unbound rule variables, they can still be created ex nihilo in hybrid advice. Similar to the logic

forward chainer, an assert expression can be branched for each binding of a rule variable. Contrary to the logic language, this does not result in multiple facts being generated and asserted, but by asserting all the objects to which the variable is bound. The hybrid aspect in code fragment 7.37 illustrates this.

```
1. ?jp targeting: #calculatePrice in: Order
2. :c
3.   | product |
5.   product ?= (c at: 1) items.
5.   product !.
6.   _proceed
```

Code Fragment 7.37: The hybrid aspect for integrating the method `calculatePrice` defined in `Order` with asserting products.

line 6 This line results in multiple assert, one for each product object to which the rule variable `product` is bound.

7.4 Summary

This chapter presents proof-of-concept implementations of three flavours of hybrid composition and hybrid aspects for integrating object-oriented and rule-based programs. The flavours correspond to three concrete rule-based languages: a Prolog extended with objects, a logic-based forward chainer and a production rule language. The object-oriented language is Smalltalk, which is a pure object-oriented programming language. Although there is some variability in the different implementations of hybrid composition on the one hand and hybrid aspects on the other, many features can be reused. The examples used in this chapter illustrate a concrete strategy for hybrid composition and a concrete syntax for hybrid aspects. This contrasts with the abstract and language-independent model of our approach in the previous chapter. We discuss in the last chapter how existing aspect-oriented technologies can replace parts of our current implementation for improving it, without this requiring fundamental changes to the underlying model (section 9.2).

Chapter 8

Evaluation

This chapter evaluates our approach against the criteria we established in section 6.2. First, we explain the case study in section 8.1. Then we illustrate the repercussions of using objects in rule-based programs in sections 8.2 and 8.3. The last four sections — 8.4, 8.5, 8.6 and8.7 — review how our approach, which combines hybrid composition and hybrid aspects, addresses the four criteria.

8.1 Case Study

8.1.1 Goal

The goal of the evaluation is to investigate if our approach fulfils the criteria we set forth earlier in section 6.2. These criteria are distilled from issues that arise when integrating rule-based and object-oriented programs and languages. These issues were uncovered when studying related work and existing systems in the first part of this dissertation, in particular in chapters 2 and 5.

In order to achieve this goal, we select a problem that is suitably complex to demonstrate the merits of our approach. We do not, however, provide a full-scale, industrial case study, since hybrid composition and hybrid aspects presented in this dissertation are in an early stage of development: the model, though complete, considers basic features of object-oriented and rule-based languages and the implementation is prototypical.

The evaluation is only performed using the instance of our approach that deals with backward-chaining logic languages, such as the Prolog-like language in OReA. The most important argument for this decision is that backward-chaining languages are more challenging to integrate with object-oriented languages than their forward-chaining counterparts. For example, the issue of returning inference results, which are bindings of rule variables, to object-oriented programs is more complex than the implicit side-effects induced by forward-chainers. A reason for selecting the logic-based formalism, is that hybrid production systems typically extend the production rule language with object-oriented language features such as message sends. Therefore, in new generation hybrid production systems, the integration between the two languages is much smoother than in logic-based hybrid systems, although less oblivious at the language level. As for the integration at the program level, our hybrid composition and hybrid aspects offer the same aspect-oriented features for achieving obliviousness at the program level, irrespective of the rule-based language. Hence we argue that the evaluation of the logic-based, backward-chaining instance of our approach, can be extrapolated to the other instances of our approach.

8.1.2 Criteria

We quickly review the criteria, first presented in section 6.2, and indicate which portion of our approach specifically addresses each of them.

universal and customised integration: hybrid composition enables a universal and automated integration of rule-based and object-oriented programs and languages, whereas hybrid aspects support highly flexible and specific customisation of the integration

symmetric obliviousness at the program level: existing aspect-oriented programming approaches render object-oriented programs oblivious to being integrated, so our contribution is mainly the obliviousness of rule-based programs, which is achieved by hybrid aspects, and partially by hybrid composition

symmetric obliviousness at the language level: both object-oriented and rule-based languages are seamlessly integrated by hybrid aspects and partially by hybrid composition, although the use of objects in rule-based languages, which is a common practice in modern-day hybrid systems, tends to reveal the object-oriented paradigm

encapsulated integration code: hybrid aspects are able to encapsulate any code that is specific to an integration, even if this integration code is a mixture of rule-based and object-oriented behaviour activation and manipulates both objects and rule variables

8.1.3 Setup

Most existing aspect-oriented programming approaches have an object-oriented base language. Although aspect-oriented programming is still relatively new, the general acceptance and awareness of its ability to ensure obliviousness of object-oriented programs is growing steadily. We demonstrated how a selection of existing aspect-oriented approaches render object-oriented programs oblivious to being integrated with business rules in chapter 2. Therefore, in this evaluation, we will concentrate on the other, and more original side of obliviousness at the program level: making rule-based programs oblivious to being integrated with object-oriented *and other* rule-based behaviour. In this regard, we select a case study that has a considerable and sufficiently complex rule-based part: an application for semi-automatic scheduling in real-world domains. This is explained in section 8.1.4.

With this scheduling application we clearly move away from business rules, which we have used as examples throughout this dissertation. Scheduling is a well-known, knowledge-intensive task, which is most suitably represented in a rule-based language [Schreiber et al., 2000]. The items being scheduled are part of the domain knowledge and represented by objects. Other functionality besides scheduling is also implemented in the object-oriented language.

In the following, we first introduce the scheduling functionality implemented in Prolog and subsequently transform it to use objects instead of facts. This first step shows the impact of having objects in rule-based programs, which is general practice in existing hybrid systems. Even when objects are only used as data-encapsulating entities from which data is retrieved by sending messages, it is clear that hybrid composition's universal and automated integration improves current approaches. The next step is to introduce some hybrid aspects for adapting the generic rule-based behaviour of the scheduler in a highly customised way. The above illustrates that both universal and customised integration is supported by our approach. In addition to this, we demonstrate that hybrid aspects achieve obliviousness of rule-based programs as well as object-oriented programs.

Obliviousness at the language level is certainly not addressed by existing aspect-oriented approaches, nor by modern-day hybrid systems. We review the language integration issues introduced in chapter 5 and demonstrate by means of the case study that they are addressed by hybrid aspects and partially by hybrid composition. The same hybrid aspects also address encapsulation of hybrid integration code.

8.1.4 Scheduling in Business Support Applications

According to [Schreiber et al., 2000], scheduling is a typical knowledge-intensive task, which, given a set of predefined items, assigns all the items to resources on time slots in a schedule. The general steps are: specify an initial schedule, select a candidate item to be assigned, select a target resource for this item on a time slot, assign item to the target resource on the time slot, evaluate the current schedule, and modify the schedule, if needed. The selection of a target resource for an item on a time slot typically depends on hard constraints, such as restrictions and precedence, as well as soft constraints, such as preferences. Additionally, a cost function evaluates the schedule.

Scheduling is suitably represented in Prolog — witness the abundance of commercial Prolog implementations that specifically target scheduling applications, such as *Meridian* (http://www.mps.com/), *Amzi!* (http://www.amzi.com/), *IFComputer* (http://www.ifcomputer.co.jp/MINERVA/) and *Sicstus* (http://www.sics.se/sicstus), to name but a few.

Scheduling can be applied to many domains, such as job shop scheduling in production plants, scheduling of courses in schools, scheduling television broadcasts, staff scheduling in hospitals, and so on. Each domain has the same basic structure, consisting of items, resources and time slots. However, the aforementioned domains also have specific characteristics which tend to be reflected in the scheduling process. As we will see later on, our initial scheduler implementation is domain-independent. Hybrid composition and hybrid aspects customise it for the domains of schools and television broadcasting.

Because we require a "multiparadigm" case study, where both object-oriented functionality and rule-based knowledge are represented, we do not consider full automatic scheduling. Rather, we envision a business support application, which *supports* certain tasks in certain domains or businesses. Scheduling is such a rule-based task, whereas creating invoices and ordering equipment and products are object-oriented tasks, for example. In the television broadcasting domain mentioned earlier, a template schedule is typically manually created by the policy makers, which has to be filled with concrete programs. Similarly, when scheduling courses, an older schedule often serves as starting point, in which courses are moved or added. In both cases, the scheduler does not have to provide an initial schedule, select a candidate item, nor evaluate the resulting schedule. Typically, the scheduler is used to infer the possible slots and available resources to which an item can be assigned. This results in the following top-level predicate:

```
?item canMoveTo: ?slot and: ?resource
```

where the variable `?item` is bound, but where both `?slot` and `?resource` can be either bound or unbound. As such, this multiway predicate can be used

1. to check if the item can be assigned to a resouce on a given slot
2. to infer the possible slots on which the item can be assigned to a given resource
3. to infer the possible resources to which an item can be assigned on a given slot

4. to infer the possible resources and corresponding slots to and on which the item can be assigned

Queries with the above predicate are activated by a client, in our case an object-oriented program, which represents items, resources and schedules, implements other tasks such as invoicing and ordering, and takes care of user interface and persistency. An example usage scenario is that the user of our application selects an item in the graphical representation of a schedule and drags it to another slot. Before the item is assigned to some resource on the new slot, our scheduler infers if this item can be moved to the new slot and what the available resources are. Another scenario is that a new item is created and has to be placed in the schedule. At his point, the scheduler infers the available resources and corresponding slots. These results are presented, again graphically, to the user so that he or she can select a suitable slot to which the item is assigned.

8.2 Our Scheduler in Prolog

We first introduce our scheduler — we refer to it as *ours* because it is not really a standard scheduler — implemented in standard Prolog. The syntax is again keyword-based because we use SOUL as implementation language. However, in this section we do not use object-oriented features of SOUL, but only rules and facts. This first version of our scheduler consists of three kinds of rules:

1. the core scheduling rules
2. facts representing items, scheduled items, slots and resources
3. a "library" of slot comparison rules (before, within, overlap)

We only present a selection of the rules in this section, in particular those that have to be adapted when introducing objects in the rules later on (section 8.3). The entire Prolog implementation of our scheduler is given in appendix C.

The top-level predicate explained in the previous section is implemented in Prolog, shown in code fragment 8.1. We review the conditions:

`initialSlot` infers the initial slots in the schedule. In our implementation, initial slots are represented as facts, e.g. `<mo,9,11> initialSlot`. Note that in the syntax of SOUL, `<` and `>` are list delimiters, hence `<mo,9,11>` is a list of three atoms: a string and two numbers. Such a list represents a slot, which consists of a day of the week, a start hour and an end hour. Note that for the sake of simplicity, we work with slots of 2 hours.

`noOverlap` infers slots that do not overlap with already scheduled items. This relatively naive, but domain-independent policy, will be customised for different domains later on in this chapter.

`availableResource:on:` infers available resources for the item on the inferred slots.

`afterPrerequisites:` infers slots that would schedule the item after its prerequisites and before items that have it as a prerequisite.

`noProhibited:` infers slots that would not schedule the item on prohibited slots.

Example queries with the `canMoveTo:and:` predicate are the following:

```
?item canMoveTo: ?slot and: ?resource  if
   ?slot initialSlot,
   ?slot noOverlap,
   ?item availableResource: ?resource on: ?slot,
   ?item afterPrerequisites: ?slot,
   ?item noProhibited: ?slot.
```

Code Fragment 8.1: The top-level scheduling rule that concludes `canMoveTo:and:`, implemented in standard Prolog.

```
graphics canMoveTo: ?x and: r4F111
```

where the item is a course named `graphics` and the resource a room named `r4F111`, and

```
cartoons canMoveTo: <tu,11,13> and: ?y
```

where the item is a television program `cartoons` and the slot a list `<tu,11,13>`. Note that the resources in this domain are tapes on which the particular program is recorded.

Scheduled items are represented by the following facts:

```
graphics scheduledItemOn: <tu,14,16> and: r4F111.
```

and

```
cartoons scheduledItemOn: <mo,14,16> and: tape123.
```

Finally, resources for items are represented by facts as well, for example:

```
r4F111 resourceFor: graphics.
```

and

```
tape123 resourceFor: cartoons.
```

These examples show that an item itself is not represented by a fact, but by an atom. This version of our scheduler does not take more information about items into account, either course items or television program items.

The rule that is used for inferring the `noOverlap` condition in the top-level predicate (code fragment 8.1), is shown in code fragment 8.2, together with a second rule, which concludes `overlap`, used by the first. In this version of our scheduler, the rule that concludes `noOverlap` infers slots that do not overlap with the scheduled items.

```
?slot noOverlap if
   not(?slot overlap).
?slot overlap  if
   ?item2 scheduledItemOn: ?slot2 and: ?resource2,
   ?slot overlap: ?slot2.
```

Code Fragment 8.2: The rule that concludes `noOverlap` and the rule that concludes `overlap`, implemented in standard Prolog.

The rules that conclude the `overlap:` predicate (two arguments), used in a condition of the `overlap` predicate (one argument), are shown in code fragment 8.3 together with an an auxiliary predicate `overlapAux:`. We show these rules because last one is the kind that is represented very succinctly in Prolog. The next section will reveal that this is not so easily achieved when objects are introduced in the rules. During the inference process, unification "destructures" the values to which `?slot1` and `?slot2` are bound and binds the resulting elements to the corresponding variables in the `overlapAux:` rule in one go.

```
?slot1 overlap: ?slot2 if
   ?slot1 overlapAux: ?slot2.
?slot1 overlap: ?slot2 if
   ?slot2 overlapAux: ?slot1.
<?day,?start1,?end1> overlapAux: <?day,?start2,?end2>  if
   smallerOrEqual(?start1,?start2),
   smaller(?start2,?end1).
```

Code Fragment 8.3: The rules that conclude `overlap:` and the auxiliary rule that concludes `overlapAux:`, implemented in standard Prolog.

8.3 Objects in Rules

Instead of representing slots, items, scheduled items and resources for items in Prolog, the hybrid version of our scheduler represents them as objects in Smalltalk. Figure 8.1 shows a rudimentary class diagram of the main classes. All classes provide accessors and mutators for their instance variables. Note that in Smalltalk it is a coding convention to give accessors and mutators the same name as the attribute, for example `day` for accessing the attibute `day` and `day:` (with one parameter) for setting the parameter as the new value for the attibute `day`. Furthermore, a schedule is also represented explicitly as an object. It refers to a collection of scheduled items and a collection of initial slots that are generated upon initialisation.

We transform the rules of our scheduler implementation in Prolog to take the aforementioned objects as arguments instead of atoms. Once more, we review the same rules as presented in the previous section in code fragments 8.1, 8.2 and 8.3. The final, transformed rules are given in appendix C.

Since facts are no longer available to be unified with conditions for binding rule variables to required values, we need to retrieve the values from the objects that replace the facts. For example, the original rule in code fragment 8.1 binds the `?slot` variable in the first condition using the `initialSlot` fact. When considering objects instead of facts, the initial slots have to be retrieved from the schedule object. Other rules also require the schedule object for similar reasons. Hence, if we adapt the original rules to accomodate this in a naive way, we get the "would-be" rule in code fragment 8.4. The required changes, although not final yet, are marked in red.

Other rules require the schedule object as well, for example the rules that conclude `noOverlapIn:` and `overlapIn:` in code fragment 8.5, which are would-be versions of the rules in code fragment 8.2. In this case, the scheduled items have to be retrieved from the schedule.

Another typical pattern in rules that bind rule variables to objects, is illustrated in the new `overlapAux:` predicate in code fragment 8.6. Recall that the original predicate in

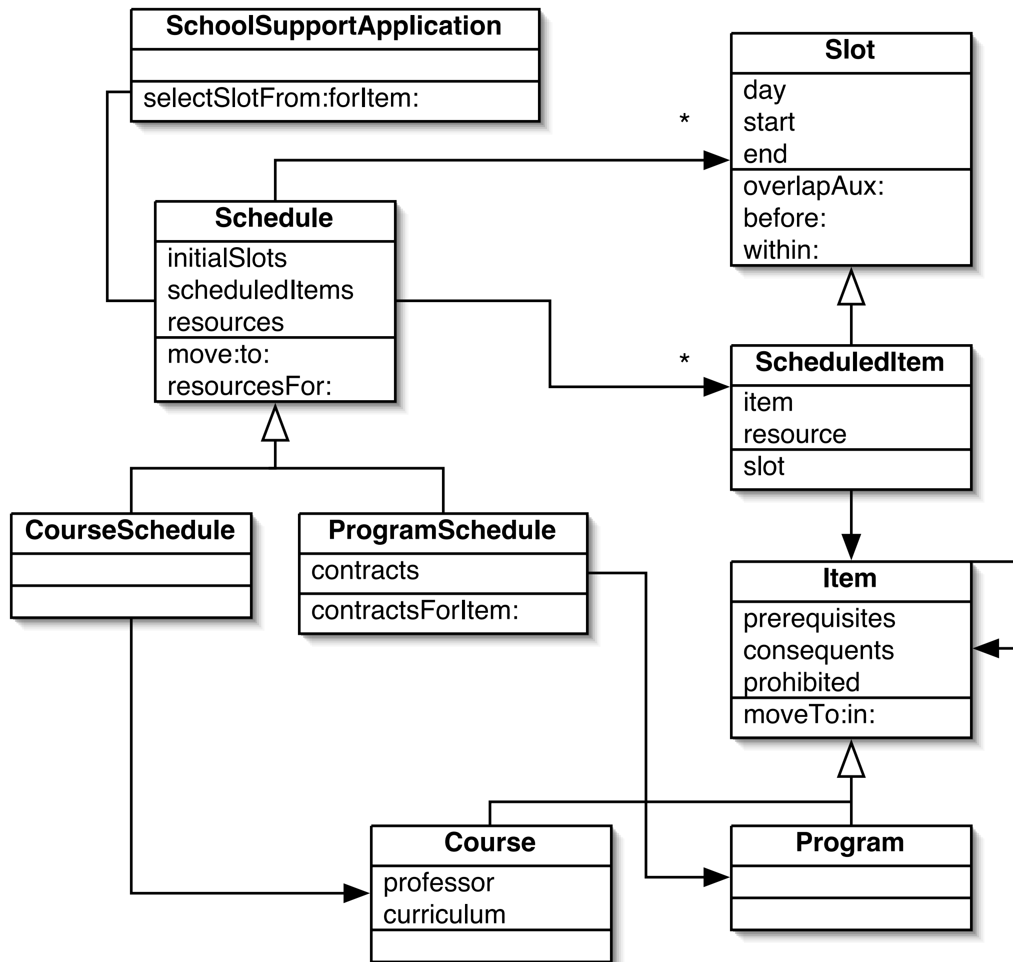


Figure 8.1: The generic domain used by our scheduler and a few specialisations of specific domains.

code fragment 8.3 destructures the values to which the variables `?slot1` and `?slot2` are bound. In the new predicate, however, the rule variables are bound to slot objects which encapsulate their state. Therefore, we need to retrieve the attributes of the slot objects explicitly, once more using the same approach as explained above. This leads to a very awkward implementation, witness code fragment 8.6. In this case, a better approach is to delegate the testing of overlap between two slots to the slot objects themselves, since this rule is typically activated with bindings for `?slot1` and `?slot2`. How to achieve this is explained in the next section.

8.4 Universal and Customised Integration

This section shows that the combination of hybrid composition and hybrid aspects achieves both universal and customised integration, a first criterion. In order to do so, we get back to the retrieval of data from objects in rules. This is typically achieved by sending messages to objects in the rules. Moreover, some subtasks of our scheduling might be more suitably represented in the object-oriented language. In this case, the subtask has to be delegated

```
?item canMoveTo: ?slot and: ?resource in: ?schedule if
   ?slot initialSlotIn: ?schedule,
   ?slot noOverlapIn: ?schedule,
   ?item availableResource: ?resource on: ?slot in: ?schedule ,
   ?item afterPrerequisites: ?slot in: ?schedule,
   ?item noProhibited: ?slot.
```

Code Fragment 8.4: A would-be top-level scheduling rule that concludes `canMoveTo:and:in:` in SOUL with objects.

```
?slot noOverlapIn: ?schedule if
   not(?slot overlapIn: ?schedule).
?slot overlapIn: ?schedule  if
   ?item2 scheduledItemOn: ?slot2 and: ?resource2 in: ?schedule,
   ?slot overlap: ?slot2.
```

Code Fragment 8.5: Would-be rules that conclude `noOverlapIn:` and `overlapIn:` in SOUL with objects.

```
?slot1 overlapAux: ?slot2  if
   ?slot1 day = ?day,
   ?slot2 day = ?day,
   ?slot1 start = ?start1,
   ?slot2 start = ?start2,
   ?slot1 end = ?end1,
   ?slot2 end = ?end2,
   smallerOrEqual(?start1,?start2),
   smaller(?start2,?end1).
```

Code Fragment 8.6: The auxiliary rule that concludes `overlapAux:`, implemented in SOUL with objects.

to an object, also achieved by sending a message to it. This is where hybrid composition comes into the picture.

A first example of the benefits of hybrid composition in our case study is by replacing the auxiliary rule that concludes `overlapAux:` in code fragment 8.6. Our implementation of hybrid composition integrates a rule-based program with an object-oriented program at a condition inference if no rule is found that concludes the condition's predicate. At such a join point, hybrid composition sends a matching message to the object to which the first argument is bound. Therefore, we remove the `overlapAux:` rule and define a method named `overlapAux:` in the class `Slot`, shown in code fragment 8.7. As mentioned earlier, when the condition with predicate `overlapAux:` is being inferred, both arguments are bound. Since the new method returns a boolean, this is automatically converted to an indication of success or failure in inferring the condition. Note that the rules in code fragment 8.3, which infer conditions with predicate `overlapAux:`, do not require to be adapted at all when the rule concluding `overlapAux:` is replaced by the method.

```
Slot >> overlapAux: aSlot
   ^self day = aSlot day
      and: [ self start <= aSlot start
      and: [ aSlot start < self end ] ]
```

Code Fragment 8.7: The method `overlapAux:` defined in the class `Slot`.

Other rules in our scheduler program that require message sending, do so because they need to retrieve data from an object. Let us consider the rule in code fragment 8.4 once more. In this code fragment, we added the extra argument for the schedule object where necessary in a naive way, denoted in red. In the first condition, the schedule object is required for retrieving the initial slots. However, this would-be condition cannot be matched automatically to a message send that corresponds to an existing method, partly because the receiver of the message has to be the schedule object to which the variable `?schedule` is bound. The `Schedule` class implements an accessors for retrieving the initial slots, `initialSlots`. Hence, the rule has to express its first condition as shown in code fragment 8.8. The result of sending the message `initialSlots` is unified with the right-hand side of the unification operator `=`. Since the result is a collection object, hybrid composition makes sure that it is interpreted as multiple bindings. When the right-hand side is an unbound variable, this results in multiple bindings of the variable. When the right-hand side is a bound variable, the unification is successful if one of the bindings unifies with the value to which the variable is bound.

```
?item canMoveTo: ?slot and: ?resource in: ?schedule  if
   ?schedule initialSlots = ?slot,
   ?slot noOverlapIn: ?schedule ,
   ?item availableResource: ?resource on: ?slot in: ?schedule ,
   ?item afterPrerequisites: ?slot in: ?schedule ,
   ?item noProhibited: ?slot.
```

Code Fragment 8.8: The real top-level rule that concludes `canMoveTo:and:`, implemented in SOUL with objects and hybrid composition.

Hybrid composition also enables the same universal and automated integration in the other direction. As an example in our case study, consider the method in code fragment 8.9. The message send with selector `canMoveTo:and:in:` is transformed in a matching query by hybrid composition because no corresponding method is defined in or inherited by the class `ScheduledItem`. This bound query results in the top-level rule of our scheduler being activated and an indication of success or failure is the result. This result is converted into the corresponding boolean value by hybrid composition and returned as if it is the result of sending the original message. In case of success, the scheduled item's slot is set to the new slot `aSlot`.

```
ScheduledItem >> moveTo: aSlot in: aSchedule
   (self canMoveTo: aSlot and: self resource in: aSchedule)
      ifTrue: [ self slot: aSlot ]
```

Code Fragment 8.9: The method `moveTo:in:` defined in the class `ScheduledItem`.

We conclude that hybrid composition provides a universal integration because it is based on a straightforward correspondence between rule-based and object-oriented program execution elements and matches identifiers of these elements unambiguously. In some cases, particularly when only a boolean or an indication of success or failure is expected from a behaviour activation, hybrid composition even achieves obliviousness at the language level.

Nevertheless, in some cases a highly customised integration is required which cannot be achieved cleanly using the abstractions a language has to offer — rules or methods for example — nor using hybrid composition. Consider the rules in code fragment 8.10, which are new versions using objects and hybrid composition of the rules in code fragment 8.2. The `noOverlapIn:` rule infers slots that do not overlap with scheduled items. However, this is a rather simplistic scheduling approach, since this prevents items from being scheduled on the same slot even if they employ different resources. However, as we see later on, in some domains this a useful approach whereas in others it is not. Integrating our scheduler as is with these different domains and their view on overlapping items, requires highly customised integration at specific points. Moreover, data from the item that is being scheduled is required, yet it is not available at these points. Hybrid aspects are able to express such customised integration.

```
?slot noOverlapIn: ?schedule if
   not(?slot overlapIn: ?schedule).
?slot overlapIn: ?schedule  if
   ?schedule scheduledItems = ?item2,
   ?item2 slot = ?slot2,
   ?slot overlap: ?slot2.
```

Code Fragment 8.10: The rule that concludes `noOverlapIn:` and the rule that concludes `overlapIn:`, implemented in SOUL with objects and hybrid composition.

In this section, we do not elaborate on the hybrid aspects that would achieve the customisations described above. The next section introduces them and shows that hybrid aspects support obliviousness at the program level. As a side effect, it becomes clear that highly customised integration is provided by hybrid aspects, in addition to universal integration provided by hybrid composition.

8.5 Symmetric Obliviousness at the Program Level

Before we present the hybrid aspects for customising the inference of overlaps, we want to stress that they especially demonstrate obliviousness of the rule-based program. We assume that achieving obliviousness of object-oriented programs is sufficiently demonstrated by existing aspect-oriented approaches, which consider an object-oriented base language, as illustrated in chapter 2. Our hybrid aspects subsume aspect-oriented programming for object-oriented languages, hence also achieves obliviousness of object-oriented programs. When this argument is not sufficient, we refer the reluctant reader to the examples provided in chapter 7.

Earlier in this chapter we indicated that we consider two domains in which our scheduler can be used: schools where courses need to be scheduled and television broadcasting where television programs need to be scheduled. First, let us consider the former domain.

```
?item canMoveTo: ?slot and: ?resource in: ?schedule  if
   ?schedule initialSlots = ?slot,
   ?slot noOverlapIn: ?schedule with: ?item,
   ?item availableResource: ?resource on: ?slot in: ?schedule ,
   ?item afterPrerequisites: ?slot in: ?schedule ,
   ?item noProhibited: ?slot.
?slot noOverlapIn: ?schedule with: ?item if
   not(?slot overlapIn: ?schedule with: ?item).
?slot overlapIn: ?schedule with: ?item  if
   ?schedule scheduledItems = ?item2,
   ?item item = ?course,
   ?item2 item = ?course2,
   ?course prof = ?prof,
   ?course curriculum = ?cur,
   or(?course2 prof = ?prof,?course2 curriculum = ?cur),
   ?item2 slot = ?slot2,
   ?slot overlap: ?slot2
```

Code Fragment 8.11: The rules concluding the predicates `canMoveTo:and:in:`, `noOverlapIn:with:` and `overlapIn:with:` after manual and invasive customisation for scheduling courses in schools.

A first customisation of inferring overlaps is required when scheduling courses in schools. In this case, a scheduled item, more specifically a course, has a professor that teaches the course and a curriculum it belongs to. A schedule contains scheduled courses for all curricula and all professors. Hence, some scheduled courses are allowed to overlap in the schedule if they are taught by different professors and provided for different curricula and therefore, students. Code fragment 8.11 marks the changes required in the original rules for accomodating the new overlap policy. Firstly, all three rules are adapted in order to make available the bindings of the `?item` variable, which is bound to a course object. Secondly, the third rule is extended with what is basically object-oriented code for retrieving professor and curriculum of a course. Finally, the same rule contains an extra condition which softens the overlap policy: overlap is only inferred when either the professor or the curriculum of the course being scheduled and an already scheduled course is the same. It is obvious that this one customisation requires many changes which crosscut the rules.

Hybrid aspects are able to encapsulate the required changes without having to adapt

the original rules in code fragments 8.8 and 8.10 to accomodate the aspects. In this case, two hybrid aspects are required: one for capturing the required course object at a join point where it is available and a second for activating the extra behaviour which is tangled in the last rule.

The first hybrid aspect is presented in code fragment 8.12. It interrupts the inference of `canMoveTo:and:in:`, at which point the course object is available. This hybrid aspect uses the `_forThisFlow:` expression with a new `SchoolOverlap` aspect instance, which is initialised with the course object from the context. This expression ensures that only one `SchoolOverlap` aspect instance is active in the control flow of each join point denoted by the `CaptureItem` aspect. This is necessary because each `SchoolOverlap` aspect instance has state — the course — that corresponds with a particular join point of `CaptureItem`. Note that the `_forThisFlow:` expression allows the original execution, i.e. inferring `canMoveTo:and:in:`, to proceed.

```
Aspect CaptureItem
?jp inferring: #canMoveTo:and:in:
:c
   _forThisFlow: (SchoolOverlap new item: (c at: 1)).
```

Code Fragment 8.12: The hybrid aspect `CaptureItem`.

The second hybrid aspect, shown in code fragment 8.13, replaces the condition inference with predicate `overlapIn:`. As such, it encapsulates all the extra behaviour that had to be added to the rule concluding `overlapIn:`, marked in red in code fragment 8.11. The `SchoolOverlap` aspect defines one aspin variable (section 6.7.4), `item`, which has been initialised by the `CaptureItem` aspect with the course being scheduled. First, one argument of the interrupted condition inference is retrieved in the advice body of the `SchoolOverlap` aspect, which is a rule variable bound to a schedule object. Then, the scheduled items are retrieved from the schedule and unified as multiple bindings with the temporary variable `item2`. The next statement, a disjunction, is branched for each binding of `item2`. It introduces the new behaviour with respect to inferring overlaps between items: overlap is only inferred when either the professor or the curriculum of the course being scheduled and an already scheduled course is the same. The last expression infers the actual overlap between the slots of the courses that are inferred by the previous expression.

```
Aspect SchoolOverlap
?jp inferring: #overlapIn:
item
:c | item2 |
   item2 ?= (c at: 2) scheduledItems.
   or(item2 content prof = item content prof,
      item2 content curriculum = item content curriculum).
   (c at: 1) ?overlap: item2 slot
```

Code Fragment 8.13: The hybrid aspect `SchoolOverlap`.

Note that the above part of our case study demonstrates that hybrid aspects are indeed able to achieve obliviousness of rule-based programs. The rule-based program does not have to be adapted at all, despite the rather challenging issue of the new behaviour requiring an

unavailable object. This program integration issue was presented in chapter 2 and referred to as a form of *dependency* (section 2.4.1). Although we only show in this chapter how this is addressed in a rule-based program, the approach is entirely similar in an object-oriented program, as well as when passing objects between rule-based and object-oriented programs. Some existing aspect-oriented approaches are able to achieve the same in an object-oriented context, particularly AspectJ, as shown in chapter 3 (section 3.5).

The second customisation of inferring overlaps in the original scheduler is in the domain of television broadcasting. As mentioned earlier, our scheduler never allows scheduled items — courses or television programs — to overlap. This is exactly the functionality that we require for the television broadcasting domain since a separate schedule is constructed for each television channel. However, television programs have contracts that specify *contract windows* in which the program is allowed to be scheduled. Basically, a contract window is represented as a slot. Therefore, a tremendous optimisation of our original scheduler is to provide the possible contract windows for a program that is being scheduled instead of all the initial slots. The hybrid aspect `TVOverlap` in code fragment 8.14 deals with this. Note that once more the program to be scheduled has to be captured first by another aspect, one almost identical to the `CaptureItem` in code fragment 8.12. The `TVOverlap` aspect denotes condition inference join points with predicate `initialSlots`, since the retrieval of initial slots from the schedule has to be replaced with the retrieval of contract slots for the program being scheduled. The aspect returns the collection of contract slots to the interrupted condition inference, which results in the collection being unified with the variable `?slot` in the first condition of code fragment 8.8.

Note, that this is an example where hybrid aspects override hybrid composition.

```
Aspect TVOverlap
?jp inferring: #initialSlots
item
:c
   (c at: 1) contractsForItem: item.
```

Code Fragment 8.14: The hybrid aspect `TVOverlap`.

8.6 Symmetric Obliviousness at the Language Level

In this section, we review the language integration issues that we identified in chapter 5 (section 5.2) and demonstrate that hybrid composition partially addresses them and hybrid aspects entirely address them, while at the same time maintaining oblviousness at the language level.

We already indicated the limits of hybrid composition with respect to language obliviousness: when only a boolean or an indication of success or failure is expected from a behaviour activation, hybrid composition enables a transparent integration. However, when other inference and message send results have to be exchanged, each base language has to be extended with features of the other base language. Yet we feel that in some cases this is acceptable, which is explained in the ensuing discussion of the language integration issues.

Hybrid aspects support integration of rule-based and object-oriented languages without each language having to be extended with features from the other. One notable exception is of course the use of objects in rules, which is acceptable because it is standard practice.

Because this issue has been explored extensively earlier in this chapter (section 8.3), we ignore it in this section.

We demonstrate the support for language obliviousness offered by hybrid composition and hybrid aspects by reviewing the language integration issues below, safe for the one concerned with objects in rules. We discuss the issues *activating rules from objects*, *returning inference results to methods* and *rule values in methods* together in section 8.6.1, whereas *sending messages from rules* and *returning message send results to rules* are presented in section 8.6.2.

8.6.1 Activating Rules from Objects

In code fragment 8.9, the bound query with predicate `canMoveTo:and:in:` is activated by hybrid composition from normal object-oriented code. Hybrid aspects can also achieve this while maintaining language obliviousness. For example, consider the method in code fragment 8.15 which simply changes the slot of an item that is being moved. The hybrid aspect in code fragment 8.16 activates the same query with predicate `canMoveTo:and:in:`, with objects taken from the interrupted execution context as arguments. If the query is succesful, the next expression is evaluated, which lets the original method execution proceed. Clearly, with such a straightforward integration, hybrid composition definitely provides the most elegant solution. However, when unanticipated and non-matching behaviour needs to be executed from an object-oriented join point, hybrid aspects are the suitable solution.

```
ScheduledItem >> moveTo: aSlot in: aSchedule
   self slot: aSlot
```

Code Fragment 8.15: An alternative method `moveTo:in:` defined in `ScheduledItem`.

```
?jp targeting: #moveTo:in: in: Schedule
:c
   (c at: 1) ?canMoveTo: (c at: 2) and: (c at: 2) resource in: (c at: 3).
   _proceed
```

Code Fragment 8.16: A hybrid aspect that checks if an item can be moved.

Consider again the above query with predicate `canMoveTo:and:in:`, but this time it has to be activated with unbound arguments from a method. When using hybrid composition, the object-oriented language has to be extended with a new kind of variable which represents rule variables, which is initially unbound and used as argument of the query. Moreover, a mechanism is required for retrieving the bindings of this rule variable when the query has finished.

Hybrid composition provides a rather transparent approach to this, more specifically interpreting uninitialised temporary variables in methods as unbound rule variables. Nevertheless, this is no standard object-oriented programming style and clearly requires the extension of the object-oriented language with a rule-based feature.

Hybrid aspects on the other hand, support declaring an unbound rule variable in the advice body and activating the query with this variable. For example, suppose a class `SchoolSupportApplication` exists which represents the application for supporting tasks in schools, such as scheduling. A user can create a new course via the graphical user interface. This results in the method `createCourse:prof:curr:` defined by this application class

being executed. At this point, the hybrid aspect given in code fragment 8.17 is executed, which activates the unbound query with the result of the original execution, two unbound rule variables, and the schedule as arguments. The last expression in the aspect is a message send which results in the inferred slots being displayed as a list from which the user can select a slot for the newly created item.

```
?jp targeting: #createCourse:prof:curr: in: SchoolSupportApplication
:c | app slot resource |
   app := c at: 1.
   item := _proceed.
   item ?canMoveTo: slot and: resource in: app schedule.
   app selectSlotFrom: slot forItem: item.
```

Code Fragment 8.17: A hybrid aspect for inferring available slots for a newly created course.

Hybrid aspects also deal transparently with converting multiple bindings for a rule variable, such as `slot` in code fragment 8.17, to a special collection object which can be treated as such by the object-oriented program or as a single item by delegating messages to the only or first element. This clearly also maintains language obliviousness.

8.6.2 Sending Messages from Rules

Again, both hybrid composition and hybrid aspects are able to cause messages being sent from rules, although in a different way.

When only an indication of success or failure is expected from a message that is sent by hybrid composition from rules, this preserves language obliviousness. This was illustrated earlier with the first two rules in code fragment 8.3, which conclude the predicate `overlap:`. They infer conditions with predicate `overlapAux:` which causes the message with the same name being sent. This in turn results in the method in code fragment 8.7 being executed. Again, hybrid aspects provide an alternative approach to achieve language obliviousness, but the solution would again be more cumbersome than the hybrid composition approach.

However, when a non-boolean result is expected from a message send, hybrid composition requires the logic-based language to be extended with a unification operator, `=` in our implementation. This obviously does not maintain obliviousness at the language level. However, in this case we feel this is acceptable, which is argued in the following. Previously, we have shown that the operator `=` can be eliminated in rules by replacing the condition, such as in the rule in code fragment 8.8. In this rule, the condition `?schedule initialSlots = ?slot` is replaced with `?schedule initialSlots: ?slot`. Two approaches can be taken to activate the desired message send in this last condition. The first approach is to define an alternative matching strategy in hybrid composition but, as we argued in section 6.6.2, this leads to a hybrid composition with too many ambiguities. Indeed, hybrid composition cannot determine automatically if a rule concluding `initialSlots:` has to be looked up, a message `initialSlots:` has to be sent or a message `initialSlots` has to be sent. The other approach is to define a hybrid aspect that interrupts the inference of `initialSlots:`, sends the message `initialSlots` in its advice, and unifies the result with the second argument of the interrupted condition, an unbound rule variable. This is exemplified by the hybrid aspect in code fragment 8.18. However, we feel that neither one of these two approaches for achieving language obliviousness outweigh the benefits of the unambiguous and automatic, albeit non-oblivious, integration provided by hybrid composition with the unification operator `=`.

```
?jp inferring: #initialSlots:
:c
   (c at: 2) ?= (c at: 1) initialSlots
```

Code Fragment 8.18: A hybrid aspect for retrieving the initial slots transparently.

Nevertheless, when messages have to be sent from a rule-based program that are entirely unanticipated and do not match at all with the rule-based join point, hybrid aspects are the most suitable solution. This is illustrated by the hybrid aspects in code fragments 8.14 and 8.17.

8.7 Encapsulated Integration Code

The `SchoolOverlap` aspect in code fragment 8.13 clearly shows that integration code can be encapsulated. Moreover, this code is decidedly hybrid: both object-oriented and rule-based behaviour is activated and both object-oriented and rule-based values are manipulated. Besides avoiding the cluttering of base programs with integration code, hybrid aspects also provide an expressive medium for representing this hybrid integration code. Indeed, although in existing hybrid systems each base language is extended with a few features of the other base language, this is often not sufficient for expressing hybrid code elegantly. This is excellently illustrated in the last rule of code fragment 8.11. The logic-based language provides the aforementioned operator `=` to accomodate the object-oriented practice of returning message send results. However, each condition can only correspond to one message send since there is no equivalent of nesting in rule-based languages. Our hybrid aspect language, on the other hand, does provide nesting of message sends in other message sends, queries and asserts, witness the much more succinct `SchoolOverlap` hybrid aspect in code fragment 8.13.

8.8 Summary

We observe that hybrid composition partly achieves obliviousness at the program level: the input programs are oblivious of what they are being integrated with. Although hybrid composition integrates input programs transparently, there has to be *something* to integrate with. More specifically, an input program must contain some statement that activates behaviour in order for hybrid composition to be able to intervene. As such, the input program must *anticipate* that a certain subtask should be carried out, although it does not need to indicate which language is going to execute the program that takes care of the subtask. Hence, if a certain subtask is not anticipated, i.e. there is no statement in the input program, hybrid composition cannot intervene. An advantage of hybrid aspects is their flexibility which allows them to intervene at any join point and activate any added or alternative behaviour, rule-based or object-oriented, without having to anticipate this in the programs being integrated.

Hybrid composition enables a universal and automatic integration of object-oriented functionality and rule-based knowledge. Hybrid aspects, on the other hand, allow customised integration and encapsulate hybrid integration code in an expressive way.

At the language level, hybrid composition encounters obliviousness problems when more sophisticated integration occurs, more specifically, when objects or rule variables are exchanged between the languages. In these cases, hybrid composition has to resort to possi-

bly ambiguous strategies for determining the correct matching behaviour activation or the languages need to be extended (with = for example) in order to bridge the paradigmatic distance and maintain linguistic symbiosis.

Contrary to hybrid composition, hybrid aspect languages act as a buffer between the two integrated languages and their paradigmatic differences. A hybrid aspect language is a symbiotic language which contains features of both integrated languages. As such, hybrid aspects incorporate all the features that had to be added to the base languages in order to enable value exchange in hybrid composition. The result is that these alien features can be altogether eliminated from the base languages. This allows total obliviousness at the language level of the input languages that are integrated by hybrid aspects.

Chapter 9

Conclusions

This chapter presents the conclusions of this dissertation. It first summarises the work presented in this dissertation while stressing our contributions (section 9.1). This dissertation ends with a discussion on future work (section 9.2).

9.1 Summary and Contributions

The goal of this dissertation is to investigate and address the lack of obliviousness at the program and the language level between rule-based knowledge and object-oriented functionality, each represented in the corresponding language of a hybrid system. We claim and endeavour to show that aspects are able to achieve the desired program and language obliviousness. As such, this work combines two seemingly unrelated fields: multiparadigm programming in hybrid systems and aspect-oriented programming. This dissertation introduces the relevant topics in a piecemeal fashion and this summary follows the same steps: first we consider the problem in an object-oriented development context, then we add existing aspect-oriented approaches to this context, subsequently we backtrack to the first step and consider hybrid systems which integrate a rule-based language with the object-oriented language, finally, all the ingredients are mixed together which results in hybrid composition and hybrid aspects.

First, the problem of integrating object-oriented functionality and rule-based knowledge is introduced, which we refer to as the lack of obliviousness at the program level. In order to illustrate this, we consider an instance of rule-based knowledge which is becoming increasingly popular nowadays: business rules. We show how separated business rules are integrated with the core functionality in a state-of-the-art, object-oriented software development context using a pattern-based approach. This approach has problems achieving obliviousness, referred to as program integration issues. These issues concern dependencies between rules and core functionality, which result in change propagation and thus complicate important software engineering tasks such as maintenance, evolution and reuse.

The next step in this dissertation consists of introducing one of the relevant fields in this familiar, object-oriented development context: aspect-oriented programming. One of the distinguishing characteristics of aspect-oriented programming approaches is ensuring obliviousness of programs being integrated. One or two exceptions aside, current aspect-oriented programming approaches support this for object-oriented programs. Therefore, a natural step we take is to apply existing aspect-oriented approaches for integrating rule-based knowledge and object-oriented functionality in an object-oriented development context. As such, we briefly review the different kinds of aspect-oriented programming approaches and provide a more thorough account of the selected ones: a composition-based and two

pointcut/advice-based approaches. This leads to the first contribution:

> **Contribution 1:** Assessment of how existing aspect-oriented programming approaches based on composition and pointcut/advice address the identified program integration issues, which concern the integration of rule-based knowledge and object-oriented functionality, in an object-oriented development context [Cibrán et al., 2003a], [Cibrán et al., 2003b].

We conclude that in an object-oriented development context, the selected aspect-oriented programming approaches make rules and core functionality oblivious of being integrated.

Since our targeted development context is hybrid systems, which combine an object-oriented programming language with a rule-based language, we introduce the latter ingredient next. In this dissertation, we use the general term rule-based language to refer to languages of the logic or production rule formalisms, which can be backward or forward chaining. Rule-based languages represent rule-based knowledge in dedicated constructs, which we refer to as rules. A rule engine controls the flow of rules automatically and provides mechanisms — explicit or implicit — for managing combinations of rules in the rule-based program itself. We conclude that the two ambassadors of rule-based languages are Prolog, a logic-based backward chainer, and OPS5, a forward chainer based on production rules.

In this step, we disregard aspect-oriented programming for a while and consider hybrid systems that enable multiparadigm programming by adding a rule-based language to the object-oriented one. This supports representation of rule-based knowledge in its own suitable paradigm instead of in the object-oriented language. We identify six issues with respect to integrating a rule-based and an object-oriented language. These issues are concerned with language support that is provided for activating rule-based programs from object-oriented programs and the other way around. This results in a second contribution:

> **Contribution 2:** A categorisation of more than 50 hybrid systems and a thorough investigation of more than a dozen of those with respect to the six language integration issues. For each of these issues, the different approaches are generalised, discussed and illustrated [D'Hondt et al., 2004].

The sheer number of hybrid systems argues in favour of our endeavour to improve integration between rule-based and object-oriented programs. Roughly half of the investigated systems employ a Prolog-like language whereas the other half bases its rule-based language on OPS5. This strengthens our previous conclusion that these rule-based languages are representative. From the survey we conclude that there are two integration philosophies which coincide with the kind of rule-based language used (Prolog-like or OPS5-like). Furthermore, we observe that the hybrid systems' approaches to addressing language integration usually do not obtain obliviousness at the language level: features inherent to the one paradigm leak through to the other and vice versa.

Before we amalgamate the previously introduced fields into an approach which applies aspect-oriented programming for achieving program and language obliviousness in hybrid systems, we establish the criteria by which potential approaches can be evaluated. We distil a set of criteria from the two sets of integration issues we identified earlier on: program and language integration issues. These criteria require fundamentally new techniques to be developed. In short, the criteria are achieving symmetric obliviousness at the program

level and the language level, supporting both automatic and customised integration, and encapsulating — possibly hybrid — integration or glue code.

The most important step in this dissertation presents our approach, a combination of hybrid composition and hybrid aspects, which applies aspect-oriented programming to hybrid systems. Since this is completely uncharted territory, our approach considers basic features of existing composition-based and pointcut/advice-based approaches. First, however, we have to consider how these approaches complement each other and how they can be combined, resulting in the next contribution:

> **Contribution 3:** A general setup for combining a composition-based and a pointcut/advice-based aspect-oriented programming approach, independently of concrete base languages. This general setup is concretised for hybrid systems, which have a rule-based and an object-oriented base language.

Secondly, we adapt each of these aspect-oriented approaches for hybrid systems, which requires determining rule-based join points in addition to the standard object-oriented join points. We argue that in the context of hybrid systems a dynamic join point model is more appropriate. This results in the following contribution:

> **Contribution 4:** A unified, dynamic join point model for pure object-oriented languages and rule-based languages, considering different formalisms and chaining strategies [D'Hondt & Jonckers, 2004]. This join point model is employed by both hybrid composition and hybrid aspects.

Although hybrid composition and hybrid aspects consider the same join point model, they integrate rule-based and object-oriented languages in a different way. Hybrid composition activates object-oriented behaviour at rule-based join points and vice versa whereas the hybrid aspects activate hybrid advice at rule-based and/or object-oriented join points. Hybrid composition requires a universal mapping between rule-based and object-oriented program (execution) elements, enabled by our next contribution:

> **Contribution 5:** Hybrid composition based on a linguistic symbiosis between pure object-oriented languages and rule-based languages, with varying rule-based formalisms and chaining strategies [D'Hondt et al., 2004].

Hybrid aspects, on the other hand, require a hybrid aspect language for expressing hybrid pointcuts and hybrid advice. In addition to the standard object-oriented pointcut designators, our hybrid aspect language provides basic rule-based pointcut designators for each kind of identified rule-based join point. Furthermore, for representing hybrid advice, our hybrid aspect language provides language features from both base languages being integrated, thus absorbing any feature of the one language leaking through to the other language.

> **Contribution 6:** Hybrid aspect languages for hybrid systems [D'Hondt & Jonckers, 2004], illustrated with an abstract grammar which takes variations with respect to rule-based formalism and chaining strategy into account.

We conclude that although hybrid composition does not attain total program and language obliviousness, it already improves the existing hybrid systems. Moreover, it provides a universal and automated mapping between rule-based and object-oriented programs. Hybrid aspects, on the other hand, have to be defined manually for each interaction between rule-based and object-oriented programs. Nevertheless, they do achieve obliviousness of both

rule-based and object-oriented programs and languages: pointcuts allow specification of rule-based and object-oriented join points without requiring invasive source code changes in either program, and the hybrid advice language absorbs any and all paradigm leaks.

After elaborating our model of hybrid composition and hybrid aspects, we provide concrete implementations for several flavours of our approach in our atelier OReA. This atelier consists of a number of useful and versatile tools for constructing different kinds of rule-based languages and corresponding flavours of hybrid composition and hybrid aspects. In particular, the basis for OReA is the state-of-the-art, object-oriented programming language Smalltalk. Other tools are the hybrid system SOUL, which implements a Prolog-like language in Smalltalk and provides some standard level of integration with it, and the aspect-oriented programming approach AspectS, which implements pointcuts and advice for Smalltalk as an object-oriented framework in Smalltalk. We use SOUL for constructing a logic-based forward chainer and a forward-chaining production system, which results in three different flavours of rule-based languages. Furthermore, AspectS is used as a starting point for implementing hybrid aspects for integrating these three rule-based languages with Smalltalk. Support for reflection in Smalltalk is used as a basis for hybrid composition, which also comes in three flavours. As expected, there is substantial commonality in the family of hybrid composition and hybrid aspect implementations. Nevertheless, a rule-based formalism and chaining strategy as well as the integration philosophy introduce variability.

> **Contribution 7:** Three different flavours of proof of concept implementations of hybrid composition and hybrid aspects in our atelier *OReA*:
>
> - for integrating Smalltalk with a Prolog-like language (logic-based and backward-chaining)
> - for integrating Smalltalk with a logic-based forward chainer
> - for integrating Smalltalk with a new generation production rule language (forward-chaining)

The final step of our work evaluates our approach of hybrid composition and hybrid aspects with respect to the criteria we set forth. This is accomplished by discussing and illustrating for each criterion how our approach addresses it. Later on in this chapter possible opportunities for future work are presented (section 9.2).

An important, further-reaching consequence of our work bears upon multilanguage programming environments in general:

> **Hybrid aspects enable a seamless integration between the languages in a multilanguage programming environment**.

There are many efforts that combine two different languages, whether they belong to similar or distinct programming paradigms. On the one hand, different languages of the same paradigm are integrated in order to benefit from characteristics of both. One example is *Agora*, which provides the dynamic programming environment of the prototype-based language Agora itself and access to the libraries of its implementation language Java [De Meuter, 1998]. A fusion of languages of different paradigms, on the other hand, allows the use of different paradigm-specific features. Hybrid systems, studied at length in this dissertation, are only one slice of the possible paradigm combinations.

In order to achieve a good integration of the desired language features, language engineers often end up designing a new language which provide a native combination of these features. However, this results in an entirely new language which just happens to combine

specific features from other languages or paradigms. As a result, there is often no backward compatibility with the languages on which the new hybrid language is based. Moreover, programmers which are adept at using one of the original languages, suddenly need to consider all the new features and how they interact with the features they are accustomed to. Furthermore, we observe that implementing the hybrid language from scratch, does not allow reuse of the original language evaluators, except perhaps copy-paste reuse of their implementations.

A consequence of this dissertation is an original and valuable approach to achieving obliviousness at the language level in multilanguage programming environments: an intermediate hybrid language is used for expressing the integration of the languages. This hybrid language consists of behaviour activating statements and dispatches behaviour activation to the correct language evaluator. Furthermore, it contains features of the original languages for manipulating and converting values that are exchanged. Although our approach also requires the design of a new hybrid language which combines features of the integrated languages, it reuses large parts of the actual evaluators of these languages. Also, there is complete backward and forward compatibility between programs written in a particular language, whether operating as a stand-alone language or integrated with another language using alternative hybrid aspects. Finally, as mentioned earlier, our approach maintains language obliviousness, which allows developers to program in their favourite language without having to deal with alien language features. We demonstrate this approach for the combination of rule-based and object-oriented languages with hybrid aspects. Yet we believe that the same general approach can be considered for other language and paradigm combinations.

9.2 Future Work

This section presents some areas in our work which can be extended or improved, and identifies several new areas closely related to this work, which can be pursued in future work. Thus, as opposed to the previous section which reviews our contributions, this section stresses what we did *not* do.

The reason for not addressing the extensions or improvements to some areas in this dissertation, is that we feel we could not contribute to the current state of the art in these areas. For each of them, we argue in the corresponding subsections that the necessary, existing techniques can be transferred without requiring fundamental adaptations to our model (section 9.2.1, 9.2.2 and 9.2.3). For each of the new areas that are outside the scope of this dissertation but are interesting to pursue, we briefly discuss the relevant issues in the corresponding subsections (section 9.2.6, 9.2.4 and 9.2.5).

9.2.1 Alternative Hybrid Composition Strategies

In chapter 6 we explain our model for hybrid composition and chapter 7 presents an implementation approach for it. Existing composition-based approaches such as HyperJ, introduced in chapter 3, provide different strategies for composing matching program elements. HyperJ has a strategy similar to our compose-by-name — referred to as *mergeByName* [Ossher & Tarr, 2001]. Contrasting with our approach, however, HyperJ also composes input programs even when both define matching elements. In this case, they are composed in the order that they are specified after the mergeByName keyword. Moreover, HyperJ supplies a strategy that does *not* compose elements when they match (*nonCorrespondingMerge*) and a strategy that overrides an element in an input program with its matching element in another input program (*overrideByName*) [Tarr & Ossher, 2000].

These alternative composition strategies can be added in a straightforward way to our model and current implementation. Indeed, HyperJ's strategies all match the program elements in the same way but take a different approach in case of a match: compose, do not compose or override. Since we provide a basic scheme for matching elements of object-oriented and rule-based program elements, referred to as linguistic symbiosis, different approaches in case of a match can be implemented as well, analogous to HyperJ.

9.2.2 Expressive Pointcut Language

In chapter 6 we introduce a model for hybrid aspects, of which an implementation is explained in chapter 7. Since many existing aspect-oriented programming approaches have a very powerful and sophisticated pointcut language, this dissertation does not focus on nor contribute to this area. Our work primarily focuses on defining a join point model for rule-based languages, and on hybrid advice, which is implemented in a hybrid language and manipulates values from rule-based and object-oriented execution contexts alike. Hybrid advice is not concerned with how the required join points are denoted by the pointcut language, only that the execution is interrupted at these join points. Advice is parametrised with values from the interrupted execution context and executed at each of these join points. One could say that we regard the pointcut language as a black box: the pointcut language considers a certain join point model and pointcut expressions denote the concrete join points at which the execution of the input programs has to be interrupted. The more elaborate the pointcut designators are with respect to qualification of join points, the more (run-time) properties can be used to distinguish between join points. Hence, a certain pointcut language can be replaced by another one as long as the same join point model is considered. We illustrate this idea in our situation in the following.

Since our current implementation is based on AspectS, it also inherits its pointcut language and adds new pointcut designators for rule-based join points that are analogous to the existing ones for object-oriented join points. However, AspectS has a very basic pointcut language: pointcut designators are minimally qualified and simply enumerated. As such, we can only distinguish between join points in a limited way: for example, an object-oriented pointcut designator captures all executions of a method with a certain name which is defined in a certain class. When a pointcut designator includes more information about a method execution, such as the number of parameters, type of the actual parameters and actual return value, the method executions that are effectively interrupted can be narrowed down. Moreover, more sophisticated pointcut designators exist, for example AspectJ's *cflow*, introduced in chapter 3, which allows picking out join points based on whether they are in a particular control flow relationship with other join points. This feature also enables more specific selection of join points. The above applies to rule-based join points as well. Important to remember is that the underlying join point model remains independent of whatever pointcut designators are introduced (such as cflow) and whatever qualification of join points is used in pointcut designators.

Another issue controlled by the pointcut language is the exposure of values from the interrupted execution context to the advice. Again, state-of-the-art pointcut languages provide mechanisms for capturing specific values from the context of a join point and exposing them to the advice. However, in AspectS, and therefore in our implementation of hybrid aspects as well, the entire interrupted execution context is passed to the advice. As a result, the necessary values need to be retrieved from this context in the advice body. This is not only cumbersome, but makes the advice very dependent on the interrupted execution context: the advice will almost certainly not be compatible with alternative interrupted execution contexts nor will it be robust enough to withstand changes to the objects in

the context. Nevertheless, replacing our current pointcut language with one that captures and exposes specific values to hybrid advice, only requires the advice to be parametrised differently, with possibly multiple specific values instead of one entire execution context. Manipulation of values from rule-based and object-oriented execution contexts remains the same.

9.2.3 Reusable Hybrid Aspects

Currently, our implementation of hybrid aspects does not support reuse of the aspects. As in AspectJ, the pointcut expression, which specifies the context in which the aspect should be applied, is tightly coupled with the advice definition, which represents the alternative or additional behaviour to be activated in the context. This results in aspects that heavily depend on the context in which they are applied and thus are hard to reuse in other contexts. Many aspect-oriented approaches exist that improve this original pointcut/advice approach in order to obtain reusable aspects and composition of aspects. *Aspectual Components* is one of the first approaches to remedy this by introducing explicit connectors that connect aspects to a specific context [Lieberherr et al., 1999]. Newer approaches, based on Aspectual Components, that pursue the same goal are *Aspectual Collaborations* [Lieberherr et al., 2003], *Caesar* [Mezini & Ostermann, 2003] and JAsCo, which is introduced in chapter 3.

Our approach can certainly be improved in this area as well. Once more, we argue that our proposed hybrid join point model and language for defining hybrid advice remains fundamentally unchanged. Let us illustrate this by considering the setup provided by JAsCo (section 3.6): an aspect consists of an advice body and one or more hooks, generic specifications of the application context of an aspect. Connectors, which are defined separately from the aspects, instantiate one or more logically related hooks with a specific context. This setup can be transferred to hybrid aspects in a straightforward way: they still consist of the same hybrid advice but have hooks instead of pointcut expressions. Since hooks are in fact abstract pointcut expressions, rule-based equivalents are easy to construe. A connector basically is a concrete pointcut expression, which can be represented in our current pointcut language, whether it is improved or not in the aforementioned areas.

9.2.4 Weaving Technologies

Although we provide proof-of-concept implementations of our model of hybrid composition and hybrid aspects in chapter 7, investigating implementation technologies for aspect-oriented programming is outside the scope of this dissertation. Our implementation can be regarded as a first prototype implementation of an interpretative weaver. This is not only a natural first step to take when implementing weavers, especially in this original application of aspect-oriented programming, but also appropriate because so many of the existing hybrid systems implement interpreters rather than compilers for their rule-based languages. Of course, this often results in an unacceptable performance overhead because the rule-base programs, the pointcut expressions and the hybrid advice are interpreted. However, this first implementation allows us to investigate the opportunities for compile-time weaving.

First of all, compile-time weaving requires determining the static counterparts of the dynamic rule-based and object-oriented join points. These *shadow* join points are used by the compile-time weaving process for inlining pointcut condition checking and advice execution. This strategy is basically the same for object-oriented and rule-based languages. Another optimisation opportunity concerns hybrid advice, which can be compiled as well.

Furthermore, we can consider hybrid systems which provide rule-based languages that are compiled to another language or an intermediate representation. Some of these systems

translate the rule-based language into the object-oriented language, or in the language interpreted by the virtual machine of the latter. Still other hybrid systems compile the rule-based language into a specialised representation, such as the *Warren Abstract Machine* [Aït-Kaci, 1991] for Prolog or a *RETE* network [Forgy, 1982] for production systems. The aforementioned compile-time weaving techniques have to be reconsidered in the context of these compiled rule-based languages.

9.2.5 Alternative Object-Oriented Languages

We attempt to keep our model of hybrid composition and hybrid aspects in chapter 6 independent from any concrete languages, rule-based as well as object-oriented. However, the implementations we provide in this dissertation of our model are for (and in) Smalltalk. There are three major differences between Smalltalk and other state-of-the-art, class-based languages such as Java and C++. First of all, Smalltalk has a pure object-oriented programming model, where everything is an object and all behaviour is achieved through message passing. In Java and C++, on the other hand, not all data are objects and some behaviour — such as for example exception handling — is not achieved with message passing. This means that our model of hybrid composition and hybrid aspects has to be extended to take these deviations from the pure object-oriented programming model into account.

The second main difference, is that Smalltalk is dynamically typed whereas Java and C++ are statically typed. In our context, this gives rise to the same, recurring "dynamic-vs-static" discussion. As we have indicated several times in chapter 6 when explaining the model of our approach, a dynamically typed language provides flexibility and transparency. This is especially useful in hybrid composition and hybrid aspects, where different, often unanticipated behaviour is activated and possibly returns values of another type. A statically typed language, on the other hand, is able to avoid type errors at run time, which are typically the result of activating alternative behaviour, because everything is checked at compile time. We observe that the difference between our current, dynamically typed implementation and a future, statically typed implementation is similar to the difference between existing weaving approaches for dynamically and statically typed base languages. For example, when in a statically typed program an aspect is weaved instead of the original behaviour, the type checker has to check at compile time whether the return type of the aspect is subsumed by the return type of the method whose execution is interrupted. Note that rule-based languages are usually not statically typed, not even the hybrid production systems that are integrated with Java.

A third point is that we only considers class-based inheritance, the reason being the obvious predominance of classes in the world of object-oriented languages today. This does however not preclude a future investigation of our results in a setting where other object classifications are possible or even desirable. The most obvious one is based on an object-type hierarchy such as the one supported by interfaces in Java. Less obvious but possibly as important, is investigating the impact of adopting an alternate inheritance strategy such as for instance prototype-based inheritance.

9.2.6 Aspect Mining and Refactoring

Another new area that can be investigated in relation to our work, is mining, documenting and refactoring of the integration points between rule-based knowledge and object-oriented functionality in hybrid systems. We have shown in this dissertation that in existing hybrid systems, rule-based and object-oriented programs activate on one another directly which

results in explicit dependencies. In order for hybrid composition and hybrid aspects to remedy this, these integration points have to be identified and transformed.

The identification of tangled code in software applications is commonly known as *aspect mining*. Although this is a relatively new area of research in the aspect-oriented community, there is already a modest body of research available. Aspect mining approaches, such as the *Aspect Browser* [Griswold et al., 2000], the *Aspect Mining Tool* [Hannemann & Kiczales, 2001], *AMTEX* [Zhang & Jacobsen, 2003] and *Concern Graphs* [Robillard & Murphy, 2002] analyse programs with respect to textual patterns, types, signatures or structure. When identifying tangled code that makes up the integration points in hybrid systems, an additional and useful indication are the paradigm leaks in the code. Both languages in current hybrid systems are typically outfitted with features of the other in order to support all the issues concerned with activation of programs in the other language. Therefore, the use of such an alien language feature in a program is an indication of the activation of a program in the other language and is a potential integration point.

In general, a part of aspect mining is that the discovered aspects are documented in a format that serves as input for refactoring. The same has to be performed for our potential aspects, i.e. the identified integration points. Refactoring enables the automatic restructuring of a program while keeping the semantics intact [Fowler, 1999]. In the context of aspect-oriented programming, research on refactoring is still embryonic. A first attempt proposes a number of aspect-oriented refactorings [Hanenberg et al., 2003]. Refactorings in the context of our work have the specific purpose of rendering integrated object-oriented functionality and rule-based knowledge in hybrid systems oblivious to one another.

Appendix A

References to the Surveyed Hybrid Systems

A.1 Surveyed Systems

A.1.1 Symbiotic Systems

- **Kiev**
  Kiev 0.9, Language Specification by Maxim Kizub, 1998
  http://forestro.com/kiev/kiev.html

- **Aion**
  Aion 9.0 Rules Guide, 2001
  Computer Associates
  http://ca.com/

A.1.2 Symmetric Systems

Logic

- **K-Prolog and its Java Interface to Prolog (JIPL)**
  http://www.kprolog.com/jipl/index_e.html

- **SICStus Prolog** and its bi-directional interface **Jasper**
  SICStus Prolog 3.11.0, User's Manual, October 2003
  Swedish Institute of Computer Science
  http://www.sics.se/isl/sicstus/docs/latest/html/sicstus.html/index.html

A.1.3 Library Systems

Production Rules

- **OPSJ**
  OPSJ 4.1, Manual by Charles L. Forgy, 2001
  Production Systems Technologies Inc.

- **Quickrules**
  QuickRules 2.5, Application Developer Manual, 2003
  YASU Technologies Inc.

- **Eclipse**
  The Haley Enterprise Inc.
  http://www.haley.com

- **Blaze**
  Developing Real-World Java Applications with Blaze Advisor, Technical White Paper, 1999
  HNC Software Inc.

- **JRules**
  JRules 4.0, Technical White Paper, 2002
  ILOG

- **Jess**
  Jess in Action, book by Ernest J. Friedman-Hill, 2003
  Sandia National Laboratories

Logic

- **SOUL**
  [Mens et al., 2001]
  http://prog.vub.ac.be/research/DMP/soul/soul2.html

- **NéOpus**
  [Pachet, 1995]

- **K-Prolog**
  http://www.kprolog.com/jipl/index_e.html

- **B-Prolog**
  B-Prolog User's Manual (Version 6.4) by Neng-Fa Zhou, 2003
  Afany Software
  http://www.probp.com/manual/index.html

- **Prolog Cafe**
  Prolog Cafe Documentation (Version 0.6), 2003
  http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/PrologCafe061/doc/index.html

- **Jinni**
  Jinni 2004 Prolog Compiler: A High Performance Java and .NET based Prolog for Object and Agent Oriented Internet Programming, USER GUIDE, 2003
  BinNet Corp.
  http://www.binnetcorp.com/Jinni/

- **InterProlog**
  InterProlog 2.03: a Java front-end and enhancement for Prolog, 2003
  Declarativa
  http://www.declarativa.com/InterProlog/

- **JavaLog**
  The Brainstorm Project
  ISISTAN Research Institute, Argentina
  http://www.exa.unicen.edu.ar/ãzunino/javalog.html

- **CommonRules**
  IBM Research
  http://www.research.ibm.com/rules/commonrules-overview.html

A.2 Catalogued Systems

A.2.1 Object-Oriented Extensions

- **LogTalk**
  http://www.logtalk.org/logtalk.html

- **Prolog++**
  [Moss, 1994]

- **Trinc Prolog**
  http://www.trinc-prolog.com/doc/pl_obj.htm

- **NUOO-Prolog**
  http://www.cs.mu.oz.au/ lee/src/oolp/

- **ALS's ObjectPro**
  Applied Logic Systems, Inc.
  http://www.als.com/alspro/objectpro.html

- **OL(P)**
  OL(P): Object Layer for Prolog – Reference&User Manual Version 1.1 for SICStus Prolog and QUINTUS Prolog by Markus P.J. Fromherz, 1993
  Xerox Corporation
  ftp://parcftp.xerox.com/pub/ol/Release-Notes

- **IF/Computer's MINERVA**
  IF Computer Software Solutions
  http://www.ifcomputer.com/MINERVA/

A.2.2 Linear Logic Languages

- **Lolli**
  http://www.lix.polytechnique.fr/Labo/Dale.Miller/lolli/

- **Lygon**
  Hitch Hikers Guide to Lygon 0.7 by Michael Winikoff, Technical Report 96/36, 1996
  Department of Computer Science, The University of Melbourne Parkville, Victoria 3052, Australia
  http://www.cs.rmit.edu.au/lygon/

- **PRObjLOG**
  PRObjLOG = PROLOG + OBJECTS implemented in Lolli, by G.Delzanno, 1997
  Department of Informatics and Computer Science, University of Genova
  http://www.disi.unige.it/person/DelzannoG/PRObjLOG/prob.html

- **YahOO**
  Objects in Forum, by G.Delzanno & M.Martelli, Proceedings of ICLP95, 1995 http://www.disi.unige.it

A.2.3 Translators

- **jProlog**
  Bart Demoen and Paul Tarau
  http://www.cs.kuleuven.ac.be/ bmd/PrologInJava

- **KLIJava**
  KLIJava: KL1 to Java Compiler based on KLIC 3.002, by Satoshi Kuramochi, 1999
  Japan Information Processing Development Center
  http://www.ueda.info.waseda.ac.jp/ satoshi/klijava/klijava-e.html

- **NetProlog**
  http://netprolog.pdc.dk

A.2.4 One-Way Hybrid Systems

- **jProlog**
  Bart Demoen and Paul Tarau
  http://www.cs.kuleuven.ac.be/ bmd/PrologInJava

- **W-Prolog**
  http://goanna.cs.rmit.edu.au/ winikoff/wp/

- **XProlog**
  http://www.iro.umontreal.ca/ vaucher/XProlog/

- **IF/Prolog**
  IF Computer Software Solutions
  http://www.ifcomputer.de/Products/Prolog/home_en.html

- **LL**
  LL-Interpreter as Cocoa-Java Application for Mac OS X, Andreas Motzek, 2001
  http://www.qrst.de/html/progsp/ll.htm

A.2.5 Rule Managment Tools

- **Business Rule Beans**
  [Rouvellou et al., 2000]

- **OptimalJ**
  CompuWare Corporation
  http://www.compuware.com/products/ optimalj/

A.2.6 Data-Change-Oriented Systems

- **Versata Logic Suite**
  Versata Inc.
  http://www.versata.com

- **USoft**
  Ness Technologies
  http://www.ness.com

- **R++**
  [Litman et al., 2002]

Appendix B

Abstract Grammar for Hybrid Aspect Languages

B.1 Common Part of the Abstract Grammar

```
<hybrid aspect definition> ::=
   ASP(<pointcut definition> , <advice definition>)

 <pointcut definition> ::=
   AND(<pointcut definition>+) |
   OR (<pointcut definition>+) |
   NOT(<pointcut definition>)  |
   <pointcut designator>

<pointcut designator> ::=
   <method execution pointcut designator> |
   <condition inference pointcut designator> |
   <conclusion generation pointcut designator>

<method execution pointcut designator> ::=
   MTH(<class name> , <method name>)

<class name> ::= <identifier>

<method name> ::= <identifier>

<condition inference pointcut designator> ::= ...

<conclusion generation pointcut designator> ::= ...

<advice definition> ::=
   ADV(<aspin variables> , <advice argument> , <temporary variables>
       , <advice statements>)

<aspin variables> ::= LST(<variable>*)

<advice argument> ::= <variable>

<temporary variables> ::= LST(<variable>*)
```

```
<variable> ::= <identifier>

<advice statements> ::= LST(<advice statement>*)

<advice statement> ::=
   OR (<advice statement>+) |
   NOT(<advice statement>)  |
   <expression>

<expression> ::=
   <imperative expression> |
   <declarative expression>

<imperative expression> ::=
   <variable> |
   <object-oriented proceed> |
   <assignment> |
   <message send>

<declarative expression> ::=
   <rule-based proceed> |
   <unification/match> |
   <query> |
   <assert>

<object-oriented proceed> ::= OPR()

<rule-based proceed> ::= RPR()

<assignment> ::= ASS(<variable> , <imperative expression>)

<unification/match> ::= UNI(<variable> , <imperative expression>)

<message send> ::=
   MSG(<receiver> , <method name> , <imperative expressions>)

<receiver> ::= <imperative expression>

<imperative expressions> ::= LST(<imperative expression>*)
```

B.2 Specific Part for Logic-Based Languages

```
<condition inference pointcut designator> ::=
   CND(<predicate name> , <predicate arity>)

<conclusion generation pointcut designator> ::=
   CNC(<predicate name> , <predicate arity>)

<predicate name> ::= <identifier>

<predicate arity> ::= <number>

<query> ::=
   QRY(<predicate name> , <imperative expressions>)

<assert> ::=
   AST(<predicate name> , <imperative expressions>)
```

B.3 Specific Part for Production Rule Languages

```
<condition inference pointcut designator> ::=
   CND(<class name> , <method name>)

<conclusion generation pointcut designator> ::=
   CNC(<class name> , <method name>)

<class name> ::= <identifier>

<method name> ::= <identifier>

<query> ::=
   QRY(<imperative expression> , <attribute name> ,
       <imperative expression>)

<assert> ::=
   AST(<imperative expression>) |
   AST(<imperative expression> , <attribute name> ,
       <imperative expression>)

<attribute name> ::=
     <identifier>
```

Appendix C

Scheduling Application

C.1 Scheduler in SOUL's Prolog

```
?item canMoveTo: ?slot and: ?resource  if
   ?slot initialSlot,
   ?slot noOverlap,
   ?item availableResource: ?resource on: ?slot,
   ?item afterPrerequisites: ?slot,
   ?item noProhibited: ?slot.

?slot noOverlap  if
   not(?slot overlap).

?slot overlap  if
   ?item2 scheduledItemOn: ?slot2 and: ?resource2,
   ?slot overlap: ?slot2.

?item availableResource: ?resource on: ?slot  if
   ?resource resourceFor: ?item,
   ?slot noOverlapResource: ?resource.

?slot noOverlapResource: ?resource  if
   not(?slot overlapResource: ?resource).

?slot overlapResource: ?resource  if
   ?item2 scheduledItemOn: ?slot2 and: ?resource2,
   ?slot overlap: ?slot2.

?item afterPrerequisites: ?slot  if
   not(?item afterConsequents: ?slot),
   not(?item beforePrerequisites: ?slot).

?item afterConsequents: ?slot  if
   ?item prerequisite: ?item2,
   ?item2 scheduledItemOn: ?slot2 and: ?resource2,
   ?slot2 before: ?slot.
```

```
?item beforePrerequisites: ?slot  if
   ?item prerequisite: ?item2,
   ?item2 scheduledItemOn: ?slot2 and: ?resource2,
   ?slot before: ?slot2.

?item noProhibited: ?slot  if
   not(?item overlapWithProhibited: ?slot).

?item overlapWithProhibited: ?slot  if
   ?item prohibited: ?slot2,
   ?slot overlap: ?slot2.

?slot1 overlap: ?slot2  if
   ?slot1 overlapAux: ?slot2.

?slot1 overlap: ?slot2  if
   ?slot2 overlapAux: ?slot1.

<?day,?start1,?end1> overlapAux: <?day,?start2,?end2>  if
   smallerOrEqual(?start1,?start2),
   smaller(?start2,?end1).

<?day1,?start1,?end1> before: <?day2,?start2,?end2>  if
   ?days days,
   ?day1 beforeAux: ?day2 in: ?days.

<?day,?start1,?end1> before: <?day,?start2,?end2>  if
   smallerOrEqual(?end1,?start2).

?d1 beforeAux: ?d2 in: <?d1|?t>  if
   !,
   member(?d2,?t).

?d1 beforeAux: ?d2 in: <?d2|?t>  if
   !,
   fail.

?d1 beforeAux: ?d2 in: <?h|?t>  if
   ?d1 beforeAux: ?d2 in: ?t.

<?day,?start1,?end1> within: <?day,?start2,?end2>  if
   smallerOrEqual(?start2,?start1),
   smallerOrEqual(?end1,?end2).

9 beginHour.
17 endHour.
<mo,tu,we,th,fr,sa,su> days.

<mo,9,11> initialSlot.
<mo,10,12> initialSlot.
<mo,11,13> initialSlot.
<mo,13,15> initialSlot.
```

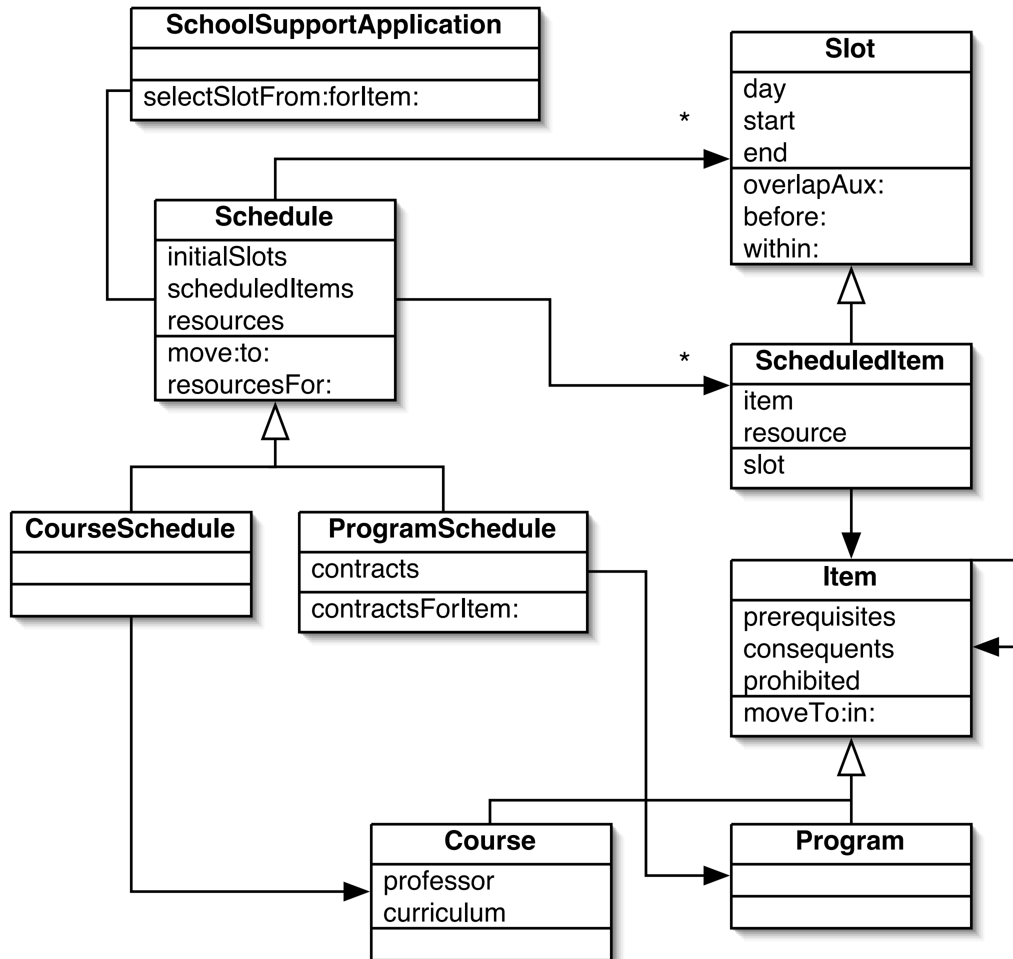
```
<mo,14,16> initialSlot.
<mo,15,17> initialSlot.
<tu,9,11> initialSlot.
...

cartoons scheduledItemOn: <mo,14,16> and: tape123.
gardening scheduledItemOn: <tu,13,15> and: tape456.

tape123 resourceFor: cartoons.
tape222 resourceFor: cartoons.
tape456 resourceFor: gardening.
...

sports prerequisite: news.
friends prohibited: <mo,9,11>.
...

graphics scheduledItemOn: <tu,14,16> and: r4F111.
distri scheduledItemOn: <we,14,16> and: r1G022.
...

r4F111 resourceFor: graphics.
r1G022 resourceFor: graphics.
r1G021 resourceFor: graphics.
r1G022 resourceFor: distri.
r1G021 resourceFor: distri.
...

distriEx prerequisite: distri.
graphics prohibited: <we,11,13>
```

C.2 Scheduler in SOUL's Prolog with Objects and Hybrid Composition

```
?item canMoveTo: ?slot and: ?resource in: ?schedule  if
   ?schedule initialSlots = ?slot,
   ?slot noOverlapIn: ?schedule ,
   ?item availableResource: ?resource on: ?slot in: ?schedule ,
   ?item afterPrerequisites: ?slot in: ?schedule ,
   ?item noProhibited: ?slot.

?slot noOverlapIn: ?schedule  if
   not(?slot overlapIn: ?schedule).

?slot overlapIn: ?schedule  if
   ?schedule scheduledItems = ?scheduledItem2,
   ?scheduledItem2 slot = ?slot2,
   ?slot overlap: ?slot2.

?item availableResource: ?resource on: ?slot in: ?schedule  if
   ?schedule resourcesForItem: ?item  = ?resource,
   ?slot noOverlapResource: ?resource in: ?schedule.
```

```
?slot noOverlapResource: ?resource in: ?schedule  if
   not(?slot overlapResource: ?resource in: ?schedule).

?slot overlapResource: ?resource in: ?schedule  if
   ?schedule scheduledItems = ?scheduledItem2,
   ?scheduledItem2 resource = ?resource,
   ?scheduledItem2 slot = ?slot2,
   ?slot overlap: ?slot2.

?item afterPrerequisites: ?slot in: ?schedule  if
   not(?item afterConsequents: ?slot in: ?schedule),
   not(?item beforePrerequisites: ?slot in: ?schedule).

?item afterConsequents: ?slot in: ?schedule  if
   ?item consequents = ?item2,
   ?schedule scheduledItems = ?scheduledItem2,
   ?scheduledItem2 item = ?item2,
   ?scheduledItem2 slot = ?slot2,
   ?slot2 before: ?slot.

?item beforePrerequisites: ?slot in: ?schedule  if
   ?scheduledItem prerequisites = ?item2,
   ?schedule scheduledItems = ?scheduledItem2,
   ?scheduledItem2 item = ?item2,
   ?scheduledItem2 slot = ?slot2,
   ?slot before: ?slot2.

?item noProhibited: ?slot  if
   not(?item overlapWithProhibited: ?slot).

?item overlapWithProhibited: ?slot  if
   ?item prohibited = ?slot2,
   ?slot overlap: ?slot2.

?slot1 overlap: ?slot2  if
   ?slot1 overlapAux: ?slot2 .

?slot1 overlap: ?slot2  if
   ?slot2 overlapAux: ?slot1.
```

C.3 Class Diagram of the Scheduler Domains



C.4 Hybrid Aspects for Customising and Calling the Scheduler

```
Aspect CaptureItem
?jp inferring: #canMoveTo:and:in:
:c
   _forThisFlow: (SchoolOverlap new item: (c at: 1)).

Aspect SchoolOverlap
?jp inferring: #overlapIn:
item
:c | scheduledItem2 |
   scheduledItem2 ?= (c at: 2) scheduledItems.
   or(scheduledItem2 item prof = scheduledItem item prof,
      scheduledItem2 item curriculum = scheduledItem item curriculum).
   (c at: 1) ?overlap: scheduledItem2 slot
```

```
Aspect CaptureItem2
?jp inferring: #canMoveTo:and:in:
:c
   _forThisFlow: (TVOverlap new item: (c at: 1)).

Aspect TVOverlap
?jp inferring: #initialSlots
item
:c
   (c at: 1) contractsForItem: item.

Aspect CallScheduler
?jp targeting: #createCourse:prof:curr: in: SchoolSupportApplication
:c | app slot resource |
   app := c at: 1.
   item := _proceed.
   item ?canMoveTo: slot and: resource in: app schedule.
   app selectSlotFrom: slot forItem: item.
```

Bibliography

Aikins, J. (1984). A representation scheme using both frames and rules. In B. Buchanan & E. Shortliffe (Eds.), *Rule-Based Expert Systems* (pp. 424–440). Readin, Mass.: Addison-Wesley. 9, 56

Aït-Kaci, H. (1991). *Warren's abstract machine: a tutorial reconstruction*. MIT Press. 166

Arsanjani, A. (2001). *Rule object 2001: A pattern language for adaptive and scalable business rule construction*. Technical report, IBM T.J. Watson Research Centre. 2, 20

Arsanjani, A. & Alpigini, J. (2001). Using grammar-oriented object design to seamlessly map business models to component-based software architectures. In *Proceedings of the International Symposium of Modelling and Simulation* (pp. 186–191). 19

Bergmans, L. & Aksit, M. (2001). Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10), 51–57. 30

BRG (2001). *Defining Business Rules: What Are They Really?* Business Rule Group, http://www.businessrulesgroup.org/. 18

Brichau, J., Gybels, K., & Wuyts, R. (2002). Towards linguistic symbiosis of an object-oriented and a logic programming language. In *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages at the European Conference on Object-Oriented Programming*. 2

Brownston, L., Farrell, R., Kant, E., & Martin, N. (1985). *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley. 44, 52

Bugliesi, M., Delzanno, G., Liquori, L., & Martelli, M. (2000). Object calculi in linear logic. *Journal of Logic and Computation*, 10(1), 75–104. 57

CACMAdaptive (2002). The adaptive web. Communications of the ACM, June. 19

Casanova, M., D'Hondt, M., & Wallet, T. (2001). Explicit domain knowledge in geographic information systems. In *Proceedings of the 14th Conference on Software Engineering and Knowledge Engineering (SEKE '01)*: Knowledge Systems Institute.

Cibrán, M. & D'Hondt, M. (2003). Composable and reusable business rules using AspectJ. In *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT) at the International Conference on Aspect-Oriented Software Development*.

Cibrán, M., Suvée, D., D'Hondt, M., & Vanderperren, W. (2004). Linking business rules with aspect-oriented programming. In *Software Engineering and Knowledge Engineering 2004 (submitted).*

Cibrán, M. A., D'Hondt, M., & Jonckers, V. (2003a). Aspect-oriented programming for connecting business rules. In *Proceedings of the 6th International Conference on Business Information Systems.* 6, 7, 24, 34, 105, 160

Cibrán, M. A., D'Hondt, M., Suvée, D., Vanderperren, W., & Jonckers, V. (2003b). Jasco for linking business rules to object-oriented software. In *Proceedings of International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'03).* 6, 7, 24, 37, 160

Date, C. (2000). *What not How: The Business Rules Approach to Application Development.* Addison-Wesley. 1, 2, 3, 56

De Meuter, W. (1998). The story of the simplest mop in the world, or, the scheme of object-orientation. *Prototype-Based Programming (eds: James Noble, Antero Taivalsaari, and Ivan Moore).* 60, 162

De Meuter, W., D'Hondt, M., Goderis, S., & D'Hondt, T. (2001). Reasoning with design knowledge for interactively supporting framework reuse. In *Proceedings of the Second International Workshop on Soft Computing Applied to Software Engineering (SCASE '01)* (pp. 31–36).

D'Hondt, E., D'Hondt, M., & D'Hondt, T. (2002a). When Turing meets Deutsch: A confrontation between classical computing and quantum computing. In *Feyerabend Workshop at the European Conference on Object-Oriented Programming (ECOOP'02).*

D'Hondt, M. (2002a). Explicit domain knowledge in software applications. In *Doctoral Symposium at the International Conference on Software Engineering.*

D'Hondt, M. (2002b). Making software knowledgeable. In *Proceedings of the 24th International Conference on Software Engineering* (pp. 735–736).: ACM.

D'Hondt, M. & Cibrán, M. (2002). Domain knowledge as an aspect in object-oriented software applications. In *Workshop on Knowledge-Based Object-Oriented Software Engineering (KBOOSE) at the European Conference on Object-Oriented Programming (ECOOP'02).*

D'Hondt, M., De Meuter, W., & Wuyts, R. (1999). Using reflective logic programming to describe domain knowledge as an aspect. In *First Symposium on Generative and Component-Based Software Engineering.* 6, 24, 115

D'Hondt, M. & D'Hondt, T. (1999). Is domain knowledge an aspect? In *ECOOP '99, Workshop on Aspect-Oriented Programming.*

D'Hondt, M. & D'Hondt, T. (2002). The tyranny of the dominant model decomposition. In *Workshop on Generative Techniques in the context of Model-Driven Architecture at the International Conference on Object-Oriented Programming, Systems, Tools and Applications.*

D'Hondt, M., Gybels, K., & Jonckers, V. (2004). Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 19th Annual ACM Symposium on Applied Computing, Special Track on Object-Oriented Programming, Languages and Systems*: ACM Press. 6, 7, 12, 55, 75, 160, 161

D'Hondt, M. & Jonckers, V. (2004). Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In *Proceedings of the Third International Conference on Aspect-Oriented Software Development*: ACM Press. 12, 75, 161

D'Hondt, M., Mens, K., & Van Paesschen, E. (2002b). Report from the ECOOP2002 workshop on knowledge-based object-oriented software engineering. In J. Hernndez & A. Moreira (Eds.), *Workshop Reader of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*.

D'Hondt, T., Gybels, K., D'Hondt, M., & Peeters, A. (2003). Linguistic symbiosis through coroutined interpretation. In *Workshop on Language Engineering for the Post-Java Era*.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall. 1

Eder, J., Kappel, G., & Schrefl, M. (1992). Coupling and cohesion in object-oriented systems. In Conference on Information and Knowledge Management. 6

Fikes, R. & Kehler, T. (1985). The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9), 904–920. 8, 56

Filman, R. E. (2001). What is aspect-oriented programming, revisited. In *Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming (ECOOP'01)*. 5, 28

Filman, R. E. & Friedman, D. P. (2000). Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, Conference on Object-Oriented Programming, Systems, Languages and Applications*. 5, 28

Flach, P. (1994). *Simply Logical: Intelligent Reasoning by Example*. John Wiley. 44, 49

Forgy, C. L. (1982). RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1), 17–37. 51, 166

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. 167

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley. 20

Griswold, W., Kato, Y., & Yuan, J. (2000). *Aspect-Browser: Tool support for managing dispersed aspects*. Technical Report CS1999-0640. 167

Grosof, B. N., Labrou, Y., & Chan, H. Y. (1999). A declarative approach to business rules in contracts: courteous logic programs in XML. In *Proceedings of the first ACM conference on Electronic commerce* (pp. 68–77).: ACM Press. 18

Gybels, K. & Brichau, J. (2003). Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development* (pp. 60–69).: ACM Press. 103

Hanenberg, S., Oberschulte, C., & Unland, R. (2003). Refactoring of aspect-oriented software. In *Lecture Notes in Computer Science: Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays (NODe 2003), Erfurt, Germany*: Springer-Verlag Heidelberg. 167

Hannemann, J. & Kiczales, G. (2001). Overcoming the prevalent decomposition of legacy code. *Workshop on Advanced Separation of Concerns*. 167

Hannemann, J. & Kiczales, G. (2002). Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA)* (pp. 161–173). 5, 24, 29

Harrison, N. B. & Coplien, J. O. (1996). Patterns of productive software organizations. *Bell Labs Technical Journal*, 1(1), 138–145. 6

Hendrix, G. G. (1979). Encoding knowledge in partitioned networks. In N. V. Findler (Ed.), *Associative Networks: Representation and Use of Knowledge by Computers* (pp. 51–92). Orlando: Academic Press. 9, 56

Hirschfeld, R. (2002). Aspects – Aspect-Oriented Programming with Squeak. In *Lecture Notes in Computer Science: Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays (NODe 2002), Erfurt, Germany* (pp. 216 – 232).: Springer-Verlag Heidelberg. 103, 119

Hürsch, W. & Lopes, C. (1995). *Separation of Concerns*. Technical report, North Eastern University. 1

Jackson, P. (1986). *Introduction to Expert Systems*. Addison-Wesley. 8, 43, 44, 56

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. (2001). An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. 4, 28, 33

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. In M. Akşit & S. Matsuoka (Eds.), *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)* (pp. 220–242).: Springer-Verlag. 3, 4, 27

Lieberherr, K., Lorenz, D., & Mezini, M. (1999). *Programming with Aspectual Components*. Technical report, Northeastern University. NU-CCS-99-01 Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html. 29, 101, 165

Lieberherr, K., Lorenz, D., & Ovlinger, J. (2003). Aspectual collaborations: Combining modules and aspects. *British Computer Society Journal (Special issue on AOP)*, 45(5), 542–565. 29, 101, 165

Lieberherr, K., Orleans, D., & Ovlinger, J. (2001). Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10), 39–41. 30

Litman, D. J., Patel-Schneider, P. F., Mishra, A., Crawford, J. M., & Dvorak, D. (2002). R++: Adding path-based rules to C++. *Knowledge and Data Engineering*, 14(3), 638–658. 173

Masini, G., Napoli, A., Colnet, D., Leonard, D., & Tombre, K. (1991). *Object-Oriented Languages*. Academic Press. 7, 56

Masuhara, H. & Kiczales, G. (2003). Modular crosscutting in aspect-oriented mechanisms. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*. 4, 79

Mens, K., Michiels, I., & Wuyts, R. (2001). Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*. 115, 170

Mens, T. & Tourwe, T. (2001). A declarative evolution framework for object-oriented design patterns. In *Proceedings of the International Conference on Software Maintenance*. 115

Mezini, M. & Ostermann, K. (2003). Conquering aspects with Caesar. In *Proceedings of 2nd International Conference on Aspect-Oriented Software Development*: ACM Press. 29, 101, 165

Minsky, M. (1975). A framework for representing knowledge. In *The Psychology of Computer Vision*. Mc Graw-Hill. 8, 56

Moss, C. (1994). *Prolog++, The Power of Object-Oriented and Logic Programming*. Addison-Wesley. 171

Nordberg III, M. E. (2001). Aspect-oriented dependency inversion. In *Workshop on Advanced Separation of Concerns, Conference on Object-Oriented Programming, Systems, Languages and Applications*. 5, 24, 28, 29

Ossher, H. & Tarr, P. (2001). Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10), 43–50. vii, xi, 7, 29, 31, 32, 33, 163

Pachet, F. (1992). *Représentation de connaissances par objets et règles : le système NéOpus*. PhD thesis, Université Paris 6. 52

Pachet, F. (1995). On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming*, 8(4), 19–24. 170

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053–1058. 1, 4, 27

Pawlak, R., Seinturier, L., Duchien, L., & Florin, G. (2001). Jac: A flexible solution for aspect-oriented programming in Java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (pp. 1–24).: Springer-Verlag. 29

Pulvermüller, E., Speck, A., D'Hondt, M., De Meuter, W., & Coplien, J. O. (2001). Report from the ECOOP'01 workshop on feature interaction in composed systems. In *Workshop Reader of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*: Springer-Verlag.

Robillard, M. & Murphy, G. (2002). Concern graphs: Finding and describing concerns. In IEEE (Ed.), *International Conference on Software Engineering (ICSE)*. 167

Ross, R. G. (2003). *Principles of the Business Rule Approach*. Addison-Wesley. 1, 2, 56

Rossi, G., Fortier, A., Cappi, J., & Schwabe, D. (2001a). Seamless personalization of e-commerce applications. In *Proceedings of the 2nd International Workshop on Conceptual Modeling Approaches for e-Business at the 20th International Conference on Conceptual Modeling*: Springer-Verlag. 21

Rossi, G., Schwabe, D., & Guimaraes, R. (2001b). Designing personalized web applications. In *World Wide Web* (pp. 275–284). 19

Rouvellou, I., Degenaro, L., Rasmus, K., Ehnebuske, D., & McKee, B. (2000). Extending business objects with business rules. In *In Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe)* (pp. 238–249). 2, 172

Russel, S. & Norvig, P. (1995). *Artificial Intelligence, A Modern Approach*. Prentice Hall. vii, 43, 116, 117

Schreiber, A. T., Akkermans, J. M., Anjewierden, A. A., de Hoog, R., Shadbolt, N. R., Van de Velde, W., & Wielinga, B. J. (2000). *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT Press. 1, 2, 8, 45, 46, 142, 143

Stefik, M., Bobrow, D. G., Mittal, S., & Conway, L. (1983). Knowledge programming in loops: Report on an experimental course. *AI Magazine*, 4(3), 3–13. 9, 56

Stevens, W., Myers, G., & Constantine, L. (1974). Structured design. *IBM Systems Journal*, 13(2), 115–139. 6, 24

Steyaert, P. (1994). *Open Design of Object Oriented Languages*. PhD thesis, Vrije Universiteit Brussel. 12, 60

Suvée, D., Vanderperren, W., & Jonckers, V. (2003). Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of 2nd International Conference on Aspect-Oriented Software Development*: ACM Press. 29, 37, 101

Tarr, P., D'Hondt, M., Bergmans, L., & Lopes, C. V. (2000). Report from the ECOOP2000 workshop on aspects and dimensions of concern: Requirements on, and challenge problems for, advanced separation of concerns. In *Workshop Reader of the 14th European Conference on Object-Oriented Programming (ECOOP '00)* (pp. 203–240).: Springer-Verlag.

Tarr, P. & Ossher, H. (2000). *Hyper/J User and Installation Manual*. Technical report, IBM Corporation. 163

Tarr, P., Ossher, H., Harrison, W., & Stanley M. Sutton, J. (1999). N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering* (pp. 107–119).: IEEE Computer Society Press / ACM Press. 4, 27, 29

von Halle, B. (2001). *Business Rules Applied*. Wiley. 1, 2, 56, 58

Wallet, T., Casanova, M., & D'Hondt, M. (2000a). Ensuring quality of geographic data with UML and OCL. In *Third International Conference on the Unified Modeling Language (<<UML >>2000)* (pp. 225–239).: Springer-Verlag.

Wallet, T., Casanova Paez, M., & D'Hondt, M. (2000b). Adaptations to OCL for ensuring quality of geographic data. In *Companion of the 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*: ACM.

Wuyts, R. (1998). Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA '98*. 115

Wuyts, R. (2001). *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel. 115

Wuyts, R. & Mens, K. (1999). Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS Europe'99*. 115

Yoder, J. & Johnson, R. (2002). The adaptive object-model architectural style. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 20

Zhang, C. & Jacobsen, H. (2003). Quantifying aspects in middleware platforms. In A. Press (Ed.), *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD)* (pp. 130–139). 167