



Behavioral Compositions in Service Oriented Architecture

Sébastien Mosser

► To cite this version:

Sébastien Mosser. Behavioral Compositions in Service Oriented Architecture. Software Engineering [cs.SE]. Université Nice Sophia Antipolis, 2010. English. NNT : . tel-00531024

HAL Id: tel-00531024

<https://theses.hal.science/tel-00531024>

Submitted on 1 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE NICE–SOPHIA ANTIPOLIS — UFR Sciences
École Doctorale de Sciences et Technologies de l'Information
et de la Communication (STIC)

T H È S E

pour obtenir le titre de
Docteur en Sciences
de l'UNIVERSITÉ de Nice–Sophia Antipolis

Discipline: Informatique

présentée et soutenue par
Sébastien MOSSER

BEHAVIORAL COMPOSITIONS IN SERVICE–ORIENTED ARCHITECTURE

Thèse dirigée par Mireille BLAY–FORNARINO et Michel RIVEILL
soutenue le 27 octobre 2010

Jury :

M.	Don BATORY	Full Professor	Rapporteur
M.	Xavier BLANC	Professeur des Universités	Rapporteur
Mme.	Mireille BLAY–FORNARINO	Professeur des Universités	Co–directrice
M.	Pierre–Alain MULLER	Professeur des Universités	Examineur
M.	Michel RIVEILL	Professeur des Universités	Co–directeur
M.	Lionel SEINTURIER	Professeur des Universités	Examineur

*“Sometimes I’ve believed as many as
six impossible things before breakfast.”*

— Lewis Carroll

Acknowledgments

A thesis **must** start by an acknowledgments chapter. Basically because it is the only chapter that will be read by more than ten persons. A thesis is obviously a scientific adventure, but also a human one, driven by meetings and discussions.

First of all, I would like to thank the members of my thesis committee. Don Batory did an *incredible* work on the manuscript, from syntactical corrections when relevant (English is far from being my mother tongue) to open research questions. And even jokes! Xavier Blanc reviews this *large* document whilst moving into a new position in another city, and found time in his busy schedule to evaluate the document from A to Z. Pierre-Alain Muller made sacrifice of his holidays to be part of the committee and play the role of examiner. Finally, Lionel Seinturier accepted to complete the committee. I guess I'm really lucky, as this committee is composed by people I admire, for both scientific and human skills. Thank you.

Then, my thanks go to my two advisors, Mireille Blay-Fornarino and Michel Riveill. Working under your direction during these three years was an exciting adventure, daily renewed. So thank you. Mireille, for showing me that computer sciences can be fun. For warning me that doing a PhD is an exhausting and hard exercise. For accepting me as your student. For showing me daily that “honesty”, “hard work” and “power of will” always pay. For all our discussions and “fights”, which finally lead to “light”. And also for bearing my grumpiness! Michel, for accepting me in your team, and funded my first research internship. For always finding the “good” question after a presentation, the one which identifies an unexpected drawback or leads to an exciting perspective. But also for setting up team seminars in awesome places, and finally for giving me advice on how to take care of my cat!

During my stay in the lab, I was part of the RAINBOW project, and then moved into the MODALIS project to follow Mireille. These two teams are really awesome, from scientific emulation to culinary contest (knowing how to cook a cake is mandatory to join these teams). The following sentences are not sorted in any ways, and cannot be exhaustive. Thank you Johan for accepting me in your newly created team. Thank you Sabine for always fixing my weird travels or administrative troubles. Thank you Diane for your culinary talents (actually, your “cannelés” are the best I've ever ate) and your heavy-tailed laws. Thank you Philippe L. for sharing my travel misfortunes. Thank you Philippe C. for your daily jokes and usual good temper. Thank you Clémentine for our (gentle) cats and dogs discussions. Thank you Mathieu for being the most quiet and comprehensive officemate I've ever met (even if you try to eject me from my office two month earlier). Thank you Hélène R. for all the outings you've organized, even if I wasn't really available. And thank you everyone else (Alban, Pierre, Javier, Tram, Filip, Ketan, Nadia) for contributing to the global team “spirit”. In the RAINBOW project, I would like to thank Anne-Marie for her sharp advice, and for showing me Mireille's door when I expressed my interest in research. Thank you Audrey, for being a model of energy and will for everyone. Thank you Christian for all the work you are doing. Thank you Christophe for being the most undecided officemate I've ever met. Thank you Nicolas and “Vinvin” for our collaboration on FAROS. And

thank you everyone else for all the little things that transform a group of people into a team.

I also hold a junior teacher position in my former engineering school. Consequently, I want to thank here my former teachers and colleagues. H  l  ne Collavizza for showing me how a teacher should be during her lectures (actually, I owe her my vocation). An also for her sharp advice when she proof-red the logical foundations of ADORE. Marc for encouraging me to set up my first course module, and silently fixing my beginner mistakes. Erick for all our discussions about teaching and functional programming. Vincent for showing me how important it is to be rigorous when setting up a lab session. Igor for his delightful vision of discrete mathematics, and for his advice on mathematical notations used in this manuscript. Claudine for her constant irony and second-degree which transforms informal chat into delectable discussions. Roger (aka Capt'ain Roro) for constantly showing us that teaching should be fun. And also for organizing boat-races around the city harbor. Anne for being the most comprehensive department headmaster I've ever met, and for showing me the "artistic" dimension of mathematics. Paul for initiating me to grammars and compilation. And for spending a full morning to explain me how trashy and stinky my first paper was (and for giving me priceless advice on scientific writing). Johnny for stopping his car each time he recognizes me walking to the bus stop. Jean-Paul for the energy he spent to teach me Fourier transform and other signal processing mechanisms (sorry for my "allergy" to signal processing). And for the incredible work he is doing with H  l  ne to lead the development of DEVINT projects (games developed by engineer students, dedicated to visually impaired children).

Research and teaching are driven by collaboration, which leads to friendships (the contrary is also possible). I really want to thank Floréal (from Montpellier) for his \TeX pertise and the useful \LaTeX hacks he gave me during the writing process. Michel (from Mulhouse) for his viral motivation and investment in the “Computer Sciences Night” contest, and for always presenting things (even complicated meta-compilation issues) with smiles and jokes. Jean-Michel (from Toulouse) for his advice on research in general and recruitment process in particular. Nicolas (from Pau) for teasing me about my future postdoctoral position in Lille, “din ch’Nord”. Brian, for trying to kill me with a couch (yes, it is possible, even if not really usual), and for being a participant of the first Sala-party (the next one will be #15!). Jane, for her shifted point of view on software engineering. And for our interdisciplinary collaboration, starting point of an awesome friendship.

Finally, I want to thank my family. First of all my parents, who initially thought that choosing Computer Sciences was not the better idea I've ever had. Thanks Dad for having read this thesis and ask questions. Thanks Mom for handling the "reception" part of the thesis defense. Thanks "Djou" (my sister) for tolerating mom during the preparation of the reception. Then, my grandparents, aunts, uncles and cousins for always accepting my "I've got work, gotta go" excuse. Special thanks to my cousins Malika and Deborah, who respectively (i) host me in NYC during the "volcano" period and (ii) proof read the kernel of this document (if the English is ugly, it is basically Deborah's fault!). Céline, who bear me and my bad-temper for already five years now. She bears the week-ends spent in my office to end an article before the conference deadline. The sleepless nights working on this manuscript. The weeks spent far from our home for conference or collaboration reasons. The last-minute demands for graphical illustration (she is an awesome graphic artist). She also red the entire manuscript, and corrects a lot of English mistakes. Thank you, for all this things and all the others that will come. And finally, a special thank to Jinx (his name is self-descriptive), my cat, as he is part of our life for one year and a half now. His daily stupidities (devouring the first hard copy of this thesis was one of those) are really fun. At least retrospectively.

Before ending this chapter, I would like to sincerely thank two professors I had during my BSc studies. They explained me lengthily and sufficiently how it should be more productive for me to quit the University and try to learn a more “handy” job (plumbing was explicitly cited during the discussion), as it was crystal clear that I was not fit for academic studies. My stupid questions were far from their own magnificence. Thank you very much for showing me exactly what I never ever want to become: someone like you.

— *Thanks.*

Contents

Preliminary Remark & Notations	1
1 Introduction	3
1.1 Context & Problematic	3
1.2 Contribution: Behavioral Composition in SOA	4
1.3 Thesis Outline	5
1.4 Running Example: the PICWEB System	5
I State of the Art	7
2 Services, Compositions & Separation of Concerns	9
2.1 Introduction	9
2.2 Genesis & Contribution Context	10
2.2.1 From Objects to Services	10
2.2.2 Service-Oriented Architectures & Infrastructures	11
2.2.3 Web Services: A Normative Environment	12
2.3 Focus on Behavioral Compositions	13
2.3.1 From Workflows	13
2.3.2 . . . To the Business-Process Execution Language (BPEL)	14
2.4 Introducing Separation of Concerns	15
2.4.1 Aspect-Oriented Programming	16
2.4.2 Feature-Oriented Programming	17
2.5 Summary & Contribution Choices	18
3 Composing Behavioral Models	21
3.1 Introduction	21
3.2 Towards a Model-Driven Approach	22
3.2.1 Models & Meta-Models	22
3.2.2 Immediate Benefits of Model-Driven Engineering	24
3.2.3 Requirements Summary & Leading Choices	24
3.3 Cutting Edge Behavioral Modeling Formalisms	25
3.3.1 Business Abstraction (BPMN)	25
3.3.2 Graphical Syntax (UML Activity Diagrams)	26
3.3.3 Execution Semantics (Protocol Modeling)	28
3.4 Model-Driven Compositions Mechanisms	28
3.4.1 Sequence Diagrams Weaving	28
3.4.2 Conflict Detection	29

3.4.3	Parallel Composition	29
3.4.4	Multi-view modeling	29
3.5	Summary & Contribution Choices	30
Towards ...		31
II The ADORE Kernel		33
4	The ADORE Meta-Model	35
4.1	Introduction	35
4.2	Coarse-Grained Description	36
4.3	ADORE: the Formal Model	39
4.3.1	Variables (\mathcal{V}) & Constants (\mathcal{C})	39
4.3.2	Activities (\mathcal{A})	40
4.3.3	Relations (\mathcal{R})	41
4.3.4	Business Processes (\mathcal{P})	43
4.3.5	Universe (\mathcal{U})	45
4.3.6	Iteration Policies (\mathcal{I}) & Blocks	46
4.4	Properties	47
4.4.1	Variables	47
4.4.2	Activities	47
4.4.3	Relations	48
4.4.4	Business Process	48
4.4.5	Iteration Policies & Block	49
5	Equipping ADORE: Syntax & Semantics	51
5.1	Introduction	51
5.2	Graphical Syntax	51
5.2.1	Representation of Activities	52
5.2.2	Representation of Relations	53
5.2.3	Iteration Policy	53
5.2.4	Process Representation	53
5.3	Execution Semantics	54
5.3.1	Activity Lifecycle Automaton	56
5.3.2	Process Execution & Activity Triggering	57
5.3.3	Iteration Policies Handling	57
5.4	From Relations to Trigger Formula (φ)	58
5.4.1	Process Entry Point & Single Predecessors	58
5.4.2	Composition of Relations	58
5.5	Iteration Policy Handling	65
6	Interacting with ADORE: Actions & Algorithms	69
6.1	Introduction	69
6.2	Logical Foundations	70
6.2.1	Equivalence (\equiv)	70
6.2.2	Substitutions (θ)	70
6.3	The ADORE Action Language	71
6.3.1	Elementary Actions: Add & Del	71
6.3.2	Action Lists, Execution & Model Consistency	73
6.3.3	Composite Actions: Replace, Unify & Discharge	74
6.4	Defining Algorithms to Interact with Models	77
6.4.1	Traceability: Algorithm Context & Symbol Generation	77
6.4.2	Illustration #1: A Cloning Algorithm	78
6.4.3	Illustration #2: Naive Process Correlation	78
6.4.4	Composition Algorithms Defined using ADORE	78

Summary	81
III Composition Algorithms	83
7 Integrating Fragments through the WEAVE Algorithm	85
7.1 Introduction	85
7.2 Algorithm Description	86
7.2.1 Principles of the WEAVE Algorithm	86
7.2.2 SIMPLEWEAVE: Integrating a single directive	92
7.2.3 WEAVE: Integrating multiples directives	94
7.3 Behavioral Interference Detection Mechanisms	96
7.3.1 Concurrency Introduction: Variable & Termination	97
7.3.2 Equivalent Activities & Directives	99
7.3.3 Intrinsic Limitation: No Semantic Interference Detection	100
7.4 Properties Associated to the WEAVE Algorithm	100
7.4.1 Order Independence & Determinism	100
7.4.2 Preservation of Paths, Execution Order and Conditions	101
7.4.3 Completeness: Variable Initialization & Process Response	102
7.5 PICWEB Illustration	103
7.5.1 Initial Artifacts: Fragments & Process	103
7.5.2 Weaving a Fragment into Another Fragment	104
7.5.3 Enhancing the getPictures Business Process	104
8 MERGE concerns around Shared Join Points	107
8.1 Introduction	107
8.2 Algorithm Motivations: “Why not simply ordering?”	108
8.2.1 Step–Wise: Order as the Keystone of Feature Compositions	108
8.2.2 Aspect–Ordering Should Not Be a Technical Answer	108
8.2.3 Ensuring a Deterministic Composition Mechanism	109
8.2.4 Identifying Shared Join Points in Collaborative Design	110
8.3 The MERGE Algorithm	110
8.3.1 Principles	110
8.3.2 Illustrating Example	111
8.3.3 Algorithm Description & Properties	112
8.4 Taming Process Design Complexity	113
8.4.1 Prerequisite: An Algorithm Invocation Pool (\mathbb{I})	113
8.4.2 Principle: Automated Reasoning on the Invocation Pool	113
8.4.3 Storing Designers Choices in a Knowledge Basis (Ξ)	114
8.5 Modeling PICWEB	114
8.5.1 The Initial PICWEB System: FLICKR [®] , PICASA [™] & Timeout	114
8.5.2 Introducing Fault Management	115
8.5.3 Step–Wise Development: Adding Cache Mechanisms	117
9 Improving Performances with the INLINE Algorithm	123
9.1 Introduction	123
9.2 Motivating Example: PICWEB Slow Execution	124
9.2.1 Implementation Benchmark: Average Invocation Time	124
9.2.2 Weights & Costs: “Let’s reason about messages”	125
9.2.3 Inlined Benchmark	126
9.3 Definition of the INLINE algorithm	127
9.3.1 Principles	127
9.3.2 Description	128
9.3.3 Intrinsic Limitations: Inlining is not a “Holy Grail”	130
9.3.4 Integration in the ADORE Development Approach	130
9.4 Using the INLINE algorithm on PICWEB	130

10 Set-Driven Concerns: the DATAFLOW Algorithm	135
10.1 Introduction	135
10.2 Motivations: Automatic Data-Driven Algorithms	136
10.2.1 Dataflow Refactoring	136
10.2.2 Array Programming	136
10.2.3 Grid-Computing: From Mashups to Scientific Workflows	136
10.3 Definition of the DATAFLOW algorithm	136
10.3.1 Running example: <i>Adder</i>	136
10.3.2 Principles	137
10.3.3 Description of the algorithm	139
10.4 Handling Real-Life Processes	142
10.4.1 Cosmetic Activities & Policy Make-Up	142
10.4.2 Towards Iteration Compositions	142
10.4.3 Integration in the ADORE Development Approach	143
10.5 Introducing Set-Concerns in PICWEB: $tag \mapsto tag^*$	143
Summary	147
 IV Applications	 149
11 A Car Crash Crisis Management System (CCCMS)	151
11.1 Introduction	151
11.2 Realizing the CCCMS	152
11.2.1 Requirements	152
11.2.2 Modeling Use Cases as Orchestrations	152
11.2.3 Modeling Extensions as Fragments	154
11.3 Zoom on Non-Functional Concerns	155
11.3.1 Modeling Non-Functional Concerns as Fragments	155
11.3.2 Discovering Fragments Target using Logical Predicates	155
11.4 Taming Orchestration Design complexity	156
11.4.1 Modeling the Complete CCCMS	156
11.4.2 MERGE to Assess Shared Join Points	157
11.4.3 WEAVE to Support Complex Behavior Definition	158
11.4.4 Interference Detection Mechanisms in Action	158
11.5 Quantitative Analysis	161
11.5.1 Coarse-Grained Structural Complexity	161
11.5.2 Entity Provenance	162
11.5.3 Cognitive Load	162
11.6 Conclusions	163
11.6.1 Approach Intrinsic Limitations	163
11.6.2 Summary	164
12 JSEDUITE, an Information Broadcasting System	167
12.1 Introduction	167
12.2 From Architectural Principles to Implementation	168
12.2.1 Information Broadcasting Candidates (IBC)	168
12.2.2 Profile & Sources of Information	168
12.2.3 Information Providers	170
12.2.4 Implementation	170
12.3 Capitalizing Designers Knowledge with ADORE	170
12.3.1 Supporting Providers & Sources Design (P_1, P'_1)	171
12.3.2 Broadcasting Policies & NF-Concerns as Fragment (P_2, P'_2)	171
12.3.3 Multiple Calls as DATAFLOW Usage	173
12.4 Conclusions	175
12.4.1 Summary	175

12.4.2 Towards an Information Broadcast Software Product Line	175
Summary	177
V Perspectives & Conclusions	179
13 Perspectives ...	181
13.1 Introduction	181
13.2 Upstream: From Requirements Engineering to ADORE	182
13.2.1 Context	182
13.2.2 Proposed Approach	182
13.2.3 Firsts Experiments	183
13.2.4 Summary	184
13.3 Downstream: Visualizing & Assessing Compositions	184
13.3.1 Context	184
13.3.2 Proposed Approach & Firsts Experimentations	184
13.3.3 Summary	186
13.4 Sketched Perspectives	186
13.4.1 Users Experiments	186
13.4.2 ADORE Expressiveness & Tooling	186
13.4.3 Actions Sequences & Optimizations	187
13.4.4 Knowledge Representation & Semantic Web	187
13.4.5 Dynamic Weaving	187
14 Conclusions	189
Appendices	193
A Implementation of ADORE	195
A.1 Tool Support	195
A.2 Composition Algorithms Invocation	197
A.3 Pointcuts	197
A.4 Fact Model & Associated Action Language	198
A.5 Execution Example	199
A.6 Execution Semantics	200
B Algorithms Implementation	201
B.1 Algorithm “Framework” & Architecture	201
B.2 Complete Example: Process Simplification	202
B.3 Chaining Algorithms	203
C Simulating Aspect-Ordering with ADORE	205
C.1 Initial Artifacts	205
C.2 Default Behavior Defined by ADORE	207
C.3 Simulating “Layered” Weaving	207
C.4 Introducing Aspect Ordering	207
C.5 ASPECTJ Implementation	211
D A BPEL implementation of PICWEB	213
D.1 Implementation Overview & Choices	213
D.2 Implementing Processes with the NETBEANS 6.7 IDE	213
D.3 XML Code Associated to PICWEB	213

E	An ASPECTJ implementation of PICWEB	221
E.1	Initial System	221
E.2	Enhancing PICWEB with ASPECTJ	222
E.2.1	Abstract Aspect to Factorize the Pointcut	222
E.2.2	Enhancements: picasa & truncate	223
E.2.3	Compilation & Execution of the Aspectized System	223
E.2.4	Interference Resolution	224
E.3	Complete Implementation of PICWEB	224
	Bibliography	227
	Abstract & Résumé	244

List of Figures

2.1	Service Brokering [Papazoglou and Heuvel, 2006]	12
3.1	BPMN formalism example: the travel booking process [White, 2005]	25
3.2	Taxonomy of UML behavioral diagrams [OMG, 2005]	26
3.3	Example of an ATM activity diagrams [Ericsonn, 2000]	27
4.1	Global overview of the ADORE meta-model	36
4.2	ADORE class hierarchy reifying activities kinds	37
4.3	ADORE class hierarchy reifying relations kinds	37
4.4	ADORE concepts associated to Variables and Constants	38
4.5	ADORE extension to integrate Fragment concept	38
4.6	“Model to Closed Term” mapping example	41
4.7	truncate business process modeled using ADORE formalism	44
5.1	Graphical notation associated to <i>interface</i> activities	52
5.2	Graphical notation associated to <i>business</i> activities	52
5.3	Graphical notation associated to <i>fragment</i> activities	53
5.4	Syntactic shortcut used to lighten graphical representation	53
5.5	Graphical representation associated to ADORE relations	53
5.6	Graphical notation associated to Iteration Policies	54
5.7	Graphical representation illustrated on PICWEB artifacts	55
5.8	Activity life-cycle transition function $\delta : Q \times \Sigma \rightarrow Q$	57
5.9	Trigger formulas associated to single-preceded activities	59
5.10	Building $\varphi(a)$: Φ and Φ_w illustration	60
5.11	building $\varphi(a')$: Φ_f and Φ_d inductive system	61
5.12	Building $\varphi(a)$: Multiple Faults handling in Φ_d	63
5.13	Building $\varphi(a)$: Φ_e Illustration	64
5.14	Composition of iteration and control-flow	66
5.15	Execution semantics associated to <i>serial</i> Iteration Policies	67
5.16	Execution semantics associated to <i>parallel</i> Iteration Policies	68
6.1	A sequence of actions, and the result of its execution	73
6.2	Illustrating the <i>replace</i> action: $\alpha = r(candidate, candidate^*, oracle)$	75
6.3	Illustrating the <i>unify</i> example: $\alpha = u(\{a, b\}, c)$	76
6.4	Illustration of the CLONEPROCESS algorithm	79
6.5	Illustrating the NAIVECORRELATION algorithm	80
7.1	WEAVE illustration: Introducing a fragment (f) inside an orchestration (o_1)	88

7.2	WEAVE illustration: Variable replacement	90
7.3	WEAVE Illustration: Handling blocks of activities as target	91
7.4	Illustrating expressiveness limitations in SIMPLEWEAVE	93
7.5	Process Simplification: a “cosmetic” need.	94
7.6	WEAVE Algorithm: Illustrating the usage of multiple weave directives	95
7.7	Introducing a non-deterministic access to a variable	98
7.8	Introducing non-deterministic termination in a process	99
7.9	Breaking the existing execution semantics through a fragment	102
7.10	Breaking the existing guards relation semantic through a fragment	102
7.11	Graphical representation of the <i>addPicasa</i> fragment	104
7.12	Graphical representation of the <i>timeout</i> fragment	104
7.13	Enhanced <i>addPicasa</i> , including the <i>timeout</i> concern	105
7.14	Composed <i>getPictures</i> , including <i>addPicasa</i> and <i>timeout</i>	106
8.1	Development approach: The MERGE algorithm, supporting the WEAVE one.	109
8.2	Illustrating the MERGE algorithm: $\text{MERGE}(\{bef, par, aft\}, \equiv, f)$	111
8.3	PICWEB merged fragment: PICASA™ & timeout concerns	116
8.4	Fault management (silent ignore) fragments defined in PICWEB	117
8.5	PICWEB merged fragment μ (including PICASA™ & timeout concerns)	118
8.6	<i>getPictures</i> business process, with timeout, PICASA™ & fault handling concerns	119
8.7	Complete version of the PICWEB <i>getPictures</i> business process	121
9.1	Average exec. time (ms) associated to <i>getPictures</i> , <i>truncate</i> and <i>flickr</i> invocations	124
9.2	Cost Model instantiated on the <i>getPictures</i> process	126
9.3	Comparing the inlined <i>getPictures</i> process performances and the original one	127
9.4	Artifacts used to illustrate the INLINE algorithm	129
9.5	Integration of the INLINE algorithm in the ADORE approach	131
9.6	Simplified implementation of PICWEB (without the key registry)	132
9.7	Inlined version of the PICWEB <i>getPictures</i> process	133
10.1	The <i>adder</i> orchestration: adds two integers stored in the memory	137
10.2	Illustrating the DATAFLOW reordering mechanisms	140
10.3	Automatically built <i>adder</i> ^(x*) process: “ <i>adder</i> for a set of <i>x</i> ”.	141
10.4	Integration of the DATAFLOW algorithm in the ADORE approach	143
10.5	Illustrating the reordering mechanisms need on the PICWEB example	144
10.6	Automatically obtained PICWEB process ($tag \mapsto tag^*$)	145
11.1	Orchestration representation for the <i>Capture Witness Report</i> use case (#2)	153
11.2	<i>RequestVideo</i> Fragment dealing with <i>Capture Witness Report</i> extension #3a	154
11.3	Non-functional fragments storing response times and errors	156
11.4	<i>CaptureWitnessReport</i> use case (main success, extensions & NF-concerns)	159
11.5	Evolution of the $ acts(p) $ indicator	161
11.6	Activities provenance in the final CCCMS	162
11.7	Cognitive load indicator (Main Success Scenario + Business Extensions)	163
12.1	Available Information Broadcasting Candidates (orange) in JSEDUITE	168
12.2	Source associated to the PICWEB Information Broadcasting Candidate	169
12.3	JSEDUITE big picture: Information Providers, Sources and IBC	170
12.4	Initial artifacts designed to support providers definition	171
12.5	Automatically built provider, aggregating <i>feedNews</i> and <i>picweb</i>	172
12.6	Automatic handling of multiple invocations in the <i>picweb</i> source	174
13.1	Overview of an AoURN \leftrightarrow ADORE process	182
13.2	PICWEB <i>getPictures</i> process described in the AoURN notation	183
13.3	Fragments dashboard instantiated on the CCCMS example.	185
A.1	ADORE editor, as an EMACS major mode	196

A.2	PROLOG facts, generated by the ADORE compiler	197
A.3	Execution semantics formulas, automatically computed by the engine	200
C.1	Initial process used to illustrate aspect ordering in ADORE	205
C.2	Fragments used to simulate aspect-ordering in ADORE	206
C.3	Result obtained with the usual ADORE MERGE mechanism	208
C.4	Result obtained with a <i>layered</i> mechanism	209
C.5	Process designed with fully-ordered fragment weaving	210
D.1	Initial version of the PICWEB process	214
D.2	Final version of the PICWEB process	215

List of Tables

2.1	Summary of expected properties and their implementation in AOP & FOP	19
4.1	From activity inheritance graph to <i>Kind</i> (closed-terms) model	40
6.1	Preconditions & Postconditions associated to the <i>add</i> action.	72
6.2	Preconditions & Postconditions associated to the <i>del</i> action.	73
11.1	Algorithms usage (& associated actions) when modeling the CCCMS	156
11.2	Fragments sets used as MERGE inputs in the CCCMS	158
12.1	Broadcasting Policies as Fragment in JSEDUITE	174
14.1	Summary of the contribution, according to PART I properties	191

List of Algorithms

1	CLONEPROCESS: $p \times p' \mapsto actions$	78
2	NAIVECORRELATION: $o_1 \times o_2 \mapsto actions$	79
3	SIMPLEWEAVE: $\omega \times p \mapsto actions$	92
4	WEAVE: $\Omega \times p \mapsto actions$	94
5	MERGE: $\Phi \times v \times f \mapsto actions$	112
6	INLINE: $act \times inl \times \beta \mapsto r$	130
7	DATAFLOW: $p \times v \mapsto actions$	142

A.1	Internal algorithms usage (automatically computed)	197
A.2	PROLOG code associated to the <code>logUpdate</code> non-functional concern	198
A.3	ADORE Meta-predicate used to automate pointcut matching	198
A.4	Computing composition directives associated to <code>logUpdate</code>	198
A.5	Implementation of the ADORE logical model as PROLOG facts	198
B.1	Illustration of algorithms architecture (WEAVE)	201
B.2	Illustration ADORE algorithm implementation: SIMPLIFY	202
B.3	Invoking an algorithm from another one: INSTANTIATE	203
C.1	Algorithm trigger by ADORE in <i>normal</i> mode	207
C.2	Algorithm triggered by ADORE in <i>layered</i> mode	207
C.3	Algorithm trigger by ADORE in <i>ordered</i> mode	207
C.4	Initial process implemented as a Java method	211
C.5	Aspects defined to implement the described fragments	211
D.1	BPEL implementation realizing the PICWEB business process	216
E.1	Java implementation of the PICWEB initial process: <i>getPictures</i>	221
E.2	Entry Point: the Main executable class	222
E.3	Common aspect used to factorize the PICWEB pointcut and the available services	222
E.4	Adding the PICASA™ repository with an aspect	223
E.5	Truncating the retrieved picture set with an aspect	223
E.6	Ordering aspects to solve undetected interference	224

Preliminary Remark & Notations

Preliminary remark:

- This document makes use of colors to illustrate compositions. Although not mandatory for its understanding, an online (or colored) version of this document will ease the reading.

Notations:

- Body text:
 - *Italic text* represents emphasized notion (e.g., citation, key concept),
 - **Bold text** represents critical information,
 - SMALL CAPITALS are used to denote existing names,
 - **Teletype** identifies code artifact.
- Formalism:
 - Uppercase letters in calligraphy style (\mathcal{L}) represent concepts defined in ADORE
 - * Orchestrations (\mathcal{O}), Variables (\mathcal{V}), Activities (\mathcal{A}), ...
 - Lower case letters (l) and singular symbols (act) refer to scalar variables
 - * Let $e \in \mathcal{E}$ a given “element”.
 - Upper case letters (L) or plural symbols ($vars$) refer to sets of variables
 - * Let $E = \{e_1, \dots, e_n\}$ a set of “elements”.
 - A^* defines a “zero or more” indications (inspired by regular expression notation)
 - * Let $e^* \in \mathcal{E}^*$, $e^* = \{e_1, \dots, e_n\}$ (used for variables or concepts, indifferently)
 - $A_{<}^*$ defines an explicitly ordered set (i.e., a list)
 - * Let $(e_1, \dots, e_n) \in \mathcal{E}_{<}^*$
- Relations:
 - The \rightsquigarrow arrow means “leads to”.

«It always seems impossible until it's done.»

Nelson Mandela

Contents

1.1 Context & Problematic	3
1.2 Contribution: Behavioral Composition in SOA	4
1.3 Thesis Outline	5
1.4 Running Example: the PICWEB System	5

1.1 Context & Problematic

Context Background. In the Service Oriented Architecture (SOA) paradigm an application is defined as an assembly of services that typically implements mission-critical business processes [OASIS, 2006a]. Services are loosely-coupled by definition, and complex services are built upon basics ones using composition mechanisms. These compositions describe how services will be orchestrated when implemented, and are typically created by business process specialists. The loose coupling methodology enables the separation of concerns and helps system evolution. Using Web Services as elementary services, and Orchestrations [Peltz, 2003] as composition mechanism, Web Service Oriented Architectures (WSOA) provide a way to implement these loosely-coupled architectures. The W3C consortium defines the following vocabulary for these artifacts in its reference glossary:

Service: “A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality [...]”

Web Services: “A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. [...]”

Orchestration: “An orchestration defines the sequence and conditions in which one Web service invokes other Web services in order to realize some useful function. I.e., an orchestration is the pattern of interactions that a Web service agent must follow in order to achieve its goal.” [W3C, 2004]

Problematic. In this context, a large part of the intrinsic complexity of an application is shifted into the definition of business processes. To perform such a task, process designers use “*composition of services*” mechanisms (e.g., orchestrations of services) and realize their use cases as message exchanges between services. A factor that contributes to the difficulty of developing complex SOA applications is the need to address multiple concerns in the same artifact. For a given process, a business expert designs its own process, which is then enriched by concerns added by the *security* expert, the *persistence* expert, the *sales* expert, . . .

This situation emphasizes the need for *separation of concerns* (SOC) mechanisms as a support to the design of complex business processes. In the SOC paradigm, concerns are defined separately, and finally assembled into a final system using composition techniques. In the context of SOA, it leads us to compose orchestrations, that is, to compose “composition of services”. Following such an approach, experts design *small* orchestrations, focused on their field of expertise. The complexity of building the final behavior which integrates all the concerns together is delegated to automatic algorithms. State-of-the-art approaches such as feature-oriented programming [Prehofer, 1997] and aspect-oriented programming [Kiczales et al., 1997] support a SOC approach.

Existing SOC approaches dedicated to SOA are code-driven. For example, the AO4BPEL approach [Charfi and Mezini, 2007] integrates aspect-oriented mechanisms into an existing orchestration language (BPEL). Working at the code level does not fit the reality of the SOA universe, where business processes can be expressed using different languages, according to different norms. Moreover, the code level is very detailed and somehow too complex to easily support the definition of mechanisms used to support business process compositions. The Model-Driven Engineering (MDE) paradigm [France and Rumpe, 2007] tames this complexity with a simple idea: instead of handling a system, try to reason on a partial view of your system (*i.e.*, a model), expressed according to the concepts you really need (*i.e.*, a meta-model) [Muller et al., 2009].

1.2 Contribution: Behavioral Composition in SOA

In this context, the contribution of this dissertation is to study the composition of business processes in the context of SOA, at the modeling level. The ultimate goal of this thesis is to support the composition of business process models, *i.e.*, to support at the model level the design of complex business processes through the composition of smaller artifacts, automatically.

We propose ADORE to fulfill this goal. ADORE means “*an Activity meta-moDel supOrting oRchestration Evolution*” and supports a compositional approach to design complex orchestrations. Models describing smaller orchestrations of services are composed to produce a model describing an orchestration of a larger set of services. The models of smaller orchestrations, called *orchestration fragments*, describe different aspects of a complex business process, where each aspect addresses a concern. Concerns can be non-orthogonal (*e.g.*, non-functional), and then required into multiple processes.

ADORE allows business process specialists to model different process aspects separately and then compose them. The support for separation of concerns helps to tame the complexity of creating and evolving models of large business processes in which many functional and non-functional concerns must be addressed. The ADORE framework consists of a formal model used to define both orchestrations and fragments. An action language is defined over this model, allowing one to interact with ADORE artifacts. Based on this action language, we define four composition algorithms used to support business process designers while they design complex processes. Composition is automated, and thus business process specialists do not need to manually compose the fragments they create. These algorithms ensure properties such as determinism or element preservation. As a consequence, designers can rely on the algorithm to automate the “technical” part of the composition, and simply focus on their business expertise. Interference detection mechanisms are implemented to automatically identify issues in the composed artifacts, and help designers to solve it.

ADORE can be viewed as a method that integrates Model Driven Development and Separation of Concerns techniques to support development of business-processes driven applications. It is important to note that ADORE provides support for modeling only business process concerns, that is, activities and their orchestrations. For example, the modeling of object structures manipulated by the activities is not supported in ADORE. In this respect, ADORE can be considered to be complementary to other SOC approaches that support the modeling of object structures (*e.g.*, meta-model composition), or services architectures (*e.g.*, ADL), but not of activities and their orchestrations.

1.3 Thesis Outline

This document is organized into five sections. In PART I, we study different *composition* approaches defined in the literature, dealing with composition of services (CHAP. 2) and composition of models (CHAP. 3). PART II is dedicated to the description of the ADORE kernel. The meta-model is described in CHAP. 4, and operationalized with an associated syntax and execution semantic in CHAP. 5. CHAP. 6 describes the action language associated to ADORE models, and how algorithm can be defined using this language to interact with existing models.

Based on this formal definition, PART III defines several composition algorithms. The first one defines how a fragment of process can be integrated into legacy processes (CHAP. 7). Then, we define how several fragments are merged when they enhance the same point in a process (CHAP. 8). We define in CHAP. 9 an algorithm used to *inline* business processes and then avoid costly data exchanges if possible. Finally, we describe in CHAP. 10 an algorithm able to transform an orchestration handling a scalar data into an orchestration handling a set of data.

We present in PART IV two systems used to validate this work. The first one is a car crash crisis management system (CHAP. 11), realized in the context of an AOM case study published in the TAOSD journal. The second one (CHAP. 12) describes the usage of ADORE to handle an information broadcasting system named JSEDUITE (deployed and daily used in several academic institutions).

Finally, PART V ends this document by exposing several perspectives of this work in CHAP. 13. CHAP. 14 concludes this by summarizing the contributions.

Appendixes. In APP. A, we describe the implementation of the ADORE framework. APP. B defines several technical algorithms used at the implementation level to make ADORE usable in *real-life* context (e.g., the definition of *generic* fragments or the cosmetic simplification of processes). APP. C illustrates how usual *aspect-oriented* ordering mechanisms can be realized in ADORE. Finally, the last two appendixes focus on the implementation of the running example: APP. D provides an implementation in the BPXL language, and APP. E focuses on an ASPECTJ implementation.

1.4 Running Example: the PICWEB System

To illustrate this work, we use a *running* example called PICWEB. This implementation was initially used as an example in a M.SC. course focused on “*Workflow & Business Processes*”, given in the POLYTECH’NICE engineering school. As a consequence, it follows SOA methodological guidelines [Papazoglou and Heuvel, 2006], positioning it as a typical usage of a SOA for experimental purposes. PICWEB is a subpart of a larger legacy system called JSEDUITE (see CHAP. 12). The JSEDUITE system aims to deliver information to several devices in academic institutions. It supports information broadcasting from academic partners (e.g., transport network, school restaurant) to several devices (e.g., user’s smart-phone, PDA, desktop, public screen).

PICWEB is a restricted version of JSEDUITE, focusing on picture management. Inspired by Web 2.0 *folksonomies* [Hotho et al., 2006], the PICWEB system relies on community-driven partners such as FLICKR® (provided by Yahoo!®) or PICASA™ (provided by Google®) to store and then retrieve available pictures. It allows a set of pictures with a given tag and up to a given threshold number to be retrieved from existing partner services.

Functional Requirements. The PICWEB system defines a very simple business process called **getPictures**. Based on a received *tag*, the process will invoke the different repositories (e.g., FLICKR® service) to retrieve pictures associated to this *tag* in the folksonomy. The different results are then interlaced to build a set of retrieved pictures. This set must be truncated to fit a user-given *threshold* before being returned to the user. Repositories must be invoked concurrently to optimize the execution time of the process.

Non-Functional Requirements. PICWEB defines the following non-functional requirements:

- *Timeout*: According to several causes (e.g., network latency, server overload), the distant repository may consume a lot of time to retrieve pictures. A *timeout* mechanism will disrupt the invocation of the associated repository when such situation occurs (i.e., after sixty seconds of wait).
- *Fault Management*: A repository may throw errors while invoking it (e.g., “invalid arguments”, “no match found”, “server overload”). These fault must be stored into a log for statistical purpose, but must not interfere with the execution of the process (i.e., a fault must be silently ignored).
- *Cache*: A simple cache mechanism will accelerate the average response time of the PICWEB process. If the process is invoked with a *tag* defined in the cache, the process must return the cached data instead of performing the computation. We assume that the temporal validity of a cached data is handled by the cache itself.

Development Scenario. The first version of PICWEB is used for prototyping purpose only and must integrate only the FLICKR[®] repository. Since this realization of the system fits the client needs, other concerns (e.g., PICASA[™] repository usage, cache) will be integrated in the process.

Technical Realization. The PICWEB example is extracted from a *real-life* application. The way it interacts with FLICKR[®] and PICASA[™] services is driven by their respective API. To simplify the example description, we create Web Services associated to these API. They kept the principles of each API, simplifying it by hiding non-relevant methods, and providing a way to define orchestration of services in this context.

- FLICKR[®]: This service defines an operation named **getPictures**. It takes as input a **tag** and a **key** (used by the server to certify the call). The result of a call contains a set of **url** associated to relevant pictures (i.e., pictures using the **tag**).
- PICASA[™]: this service defines an **explore** operation. Based on a **tag**, it returns a set of **url** associated to relevant pictures. There is no need for a key or authentication mechanisms in the API provided by Google (at least for folksonomy exploration purpose).

PICWEB is a subpart of JSEDUITE, and uses several technical services defined in JSEDUITE:

- *Registry*: it is usual for service providers to protect their operations with the usage of a user-specific key (see FLICKR[®]). As a consequence, the key registry service can store such information, and the invocation of the **getByKey** operation returns the **key** associated to the **serviceName** received as input.
- *Cache Service*: this service allows one to introduce cache mechanisms in a business process. This feature is important to ensure short response time for users while broadcasting information. For a given **key**, the service can check the validity of the associated **data** (operation **isValid**), or return the **data** to the user (operation **get**). One can store a **data** under a given **key** through the **store** operation.
- *Stopwatch*: this service is defined to serve a statistical purpose. It allows one to retrieve the current time, and then compute execution time for a given process. In the context of PICWEB, we focus on the **wait** operation which realizes a count down mechanism: invoking **wait(4)** basically do nothing but takes four seconds to terminate.
- *Log*: this service realizes a remote logger through the definition of a **write** operation. This operation takes as input a **category** (e.g., “info”, “fault”) and a **message**.

Part I

State of the Art

Notice: *This chapter describes the “context” of the thesis contribution. We made the choice to only give a big picture of composition mechanisms. We study in CHAP. 2 mechanisms defined to handle composition of services & separation of concerns, and in CHAP. 3 mechanisms defined to handle composition of models. We postpone “specific” analysis of related works in the associated chapter.*

Services, Compositions & Separation of Concerns

«But if I try to make sense of this mess I'm in, I'm not sure where I should begin. I'm falling.»

Sum41 (“Over my head”)

Contents

2.1 Introduction	9
2.2 Genesis & Contribution Context	10
2.2.1 From Objects to Services	10
2.2.2 Service-Oriented Architectures & Infrastructures	11
2.2.3 Web Services: A Normative Environment	12
2.3 Focus on Behavioral Compositions	13
2.3.1 From Workflows ...	13
2.3.2 ... To the Business-Process Execution Language (BPEL)	14
2.4 Introducing Separation of Concerns	15
2.4.1 Aspect-Oriented Programming	16
2.4.2 Feature-Oriented Programming	17
2.5 Summary & Contribution Choices	18

2.1 Introduction

This dissertation describes a contribution to the design of Services Oriented Architecture (SOA), using composition mechanisms inspired by the Separation of Concerns (SOC) paradigm. The ultimate objective of the contribution is to tame the design of complex business processes, through a compositional approach. In this chapter, we focus on the context of this work, that is, SOA & SOC. Considering “composition” as the keystone of our contribution, we will use the successive definitions of this word to act as an Ariadne’s thread (denoted as “ \Rightarrow ”). We try here to understand how the “composition of objects” notion born twenty years ago leads us to the “composition of composition of services” problematic proposed in this dissertation. Based on these definitions, we identify several properties required by a platform which aims to tackle this issue: (i) SOA-driven properties are denoted as P_x , and (ii) SOC-driven properties are denoted as Π_x .

This chapter is organized as the following. We describe in SEC. 2.2 the genesis of SOA, and focus on behavior in SEC. 2.3. Then, we describe in SEC. 2.4 the Separation of Concerns (SOC) paradigm and its relations to the topic. We conclude this chapter by summarizing this state-of-the-art study in SEC. 2.5.

2.2 Genesis & Contribution Context

2.2.1 From Objects to Services

Objects. The *Object-Oriented* paradigm (OOP) defines useful concepts for developers, *e.g.*, *abstraction*, *encapsulation*, *modularity*. It was a major revolution in the software engineering field [Booch, 1986]. In this context, the *composition* of objects is defined as a **way to combine simple objects into more complex ones**, at a structural level. This structural operation is not specific to the OOP, but supports in this context notions such as *encapsulation* and *modularity*. At the behavioral level, compositions are structured by inheritance [Snyder, 1986], traits [Schärli et al., 2003] or mixins [Bracha and Cook, 1990] mechanisms. But composition can also be expressed directly through object references in method bodies. These fine-grained mechanisms often lead to the *spaghetti-code* anti-pattern [Brown et al., 1998], which identifies non-maintainable pieces of software. Based on *objects*, design patterns were defined as established and reusable solutions to common problems [Gamma et al., 1995]. The object composition mechanisms are highly used in these patterns, showing its importance in an object-oriented approach. These patterns were one of the initial steps to more coarse-grained software design approaches, which lead to the advent of the “component” age.

Components. In front of objects limitations in terms of granularity (limited to the fine-grained description of system, at the class or the prototype [Ungar and Smith, 1987] level), the research community produces a new paradigm, based on the *component* keystone [Szyperski, 1998]. The Component Based Software Engineering (CBSE) paradigm [Kozaczynski and Booch, 1998] relies on the definition of components, which can be seen as coarse-grained objects with explicit interfaces and then avoid the definition of spaghetti code. A component holds its contract, which describes required and provided operation. At this level of abstraction, the composition is seen as **the assembly of components to build a more complex application**. First classes entities such as connectors [Yellin and Strom, 1997] are used to realize connections between provided and required operations. These connectors support communication abstraction, and can be implemented as components [Matougui and Beugnard, 2005]. Component models allow the interchangeability of components and supports the so-called “component-off-the-shelf” approach [Medvidovic et al., 1997]. Complex applications are built as the assembly of reusable and interchangeable components [Batory and O'Malley, 1992]. The use of Architecture Description Languages (ADL, [Medvidovic and Taylor, 2000]) extends this approach and allows to formally analyze a component based architecture. These languages support the definition of high-level properties expected in the component assembly, and can be then visualized or verified [Oquendo, 2004]. The ADL domain supports formal notations in order to define structure and behavior of software architectures. For instance in [Barais and Duchien, 2005] authors use OCL to define invariant properties for component architectures and FSP Language to specify external component behaviors. This framework allows to partially check the architecture type consistency at structural and behavioral level. Several component-based models and their associated platforms are available (*e.g.*, the CORBA component model [OMG, 2006b], FRACTAL [Bruneton et al., 2004]).

Services. Where components focus on architectural description to provide reusable pieces of software, *services* focus on a business-oriented vision. In the *service-oriented* paradigm, coarse grained “business” functionality are exposed through services. The key idea is not the reusability in itself, but the definition of loosely-coupled entities [Papazoglou, 2003], specified in business terms instead of technical ones. Services are published by a provider, and discovered over the network by potential consumers. Consumers can then use the service as a black box, through its publicly available interface, without any implementation concern. In this context, the *composition* word is used to denote **the composition of service invocations to perform a business-oriented task**.

“Services reflect a “service-oriented” approach to programming, based on the idea of describing available computational resources, e.g., application programs and information system components, as services that can be delivered through a standard and well-defined interface. [...] Service-based applications can be developed by discovering, invoking, and composing network-available services rather than building new applications.” [Papazoglou, 2008]

► The meaning of “composition” slowly shifts in twenty years. From a fine grained data structure combination at the object level, the word refers to the structural assembly of components at the architectural level in the CBSE paradigm. At the service level, composition refers to a business behavior, which assembles (beyond a single program and language) invocations of several services to perform a task. This “vision” is used as the starting line of the contribution described in this dissertation.

2.2.2 Service-Oriented Architectures & Infrastructures

We focus now on the *service* paradigm, and the associated architectures built according to its principles.

Definition. Service-Oriented Architectures (SOA) were defined to support the service approach. However, there is no real consensus on the definition of the SOA concept.

- “A service-oriented architecture is a style of multi-tier computing that helps organizations share logic and data among multiple applications and usage modes” (definition published by the Gartner Group [Schulte and Natis, 1996]).
- “Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations” [OASIS, 2006a].
- “SOA enables flexible integration of applications and resources by: (1) representing every application or resource as a service with standardized interface, (2) enabling the service to exchange structured information(messages, documents, “business objects”), and (3) coordinating and mediating between the services to ensure they can be invoked, used and changed effectively” [Dodani, 2004].

In practice, the different definitions converge on the following interaction protocol: services are published by a service provider, and registered into a service directory. On the customer side, one will start by exploring the service directory to find the relevant service, and then invoke it, as depicted in FIG. 2.1.

The SOA paradigm takes its root in the previously described paradigm. The main concepts are reused, e.g., encapsulation, interchangeability, abstraction or architectural reasoning. In addition, several properties associated to services raised in the literature [Wang and Fung, 2004]. These properties address (i) implementation independence, (e.g., a client can consume indifferently a JAVA or a .NET service) (ii) state-less nature (i.e., a query is sufficient to itself to compute a result), (iii) business cohesion (i.e., exposed operations are business-related) and finally (iv) operation idempotence. These principles facilitate the definition of loosely coupled architectures.

Loosely-coupled & Composition. The key idea is to support the re-usability of preexisting pieces of software, exposed as services with the minimum amount of inter-dependencies. The business-driven “composition” of service is the keystone of the SOA paradigm. Services are published as elementary bricks one needs to compose to build business-oriented applications. The compositions outsource these dependencies out of the services, and then avoid the *spaghetti-architecture* anti-pattern [Brown et al., 1998]. This paradigm also eases the integration of business partners services, kept separated from the internal information system, and only composed

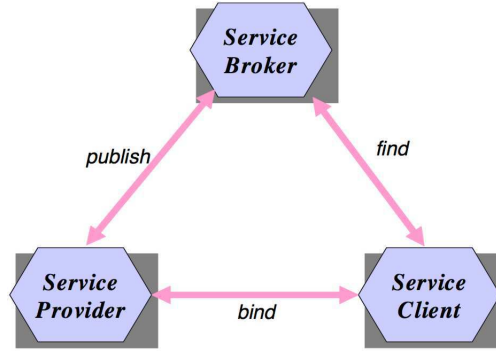


Figure 2.1: Service Brokering [Papazoglou and Heuvel, 2006]

when needed, without strong links and dependencies. As a consequence, the business process design activity shifts from developers to business experts. It leads us to the first property expected by a platform supporting the design of services compositions.

$\rightsquigarrow P_1$: **Business expressiveness.** The abstraction level associated to the definition of service compositions must reach the “business” layer.

2.2.3 Web Services: A Normative Environment

The Web Services standard can be used to implement operational SOA, known under the name of “Web Service–Oriented Architecture” (WSOA) [Peltz, 2003]. This technology is massively used in the industry, as an integration platform for legacy pieces of software [Linthicum, 2000], potentially heterogeneous. We restrict our study to *contractualized* web services (as descendants of component–based architecture), and do not address REST¹ architectures [Fielding, 2000]. Web services rely on four technologies to support the implementation of SOA: (i) XSD to describe data structures, (ii) WSDL to describe service interfaces, (iii) SOAP to exchange messages and (iv) UDDI to support service discovery. These XML–based technologies allow the definition of code generator, used to automatically address the published services.

- **XSD** The Xml Schema Description language [W3C, 2008] is a XML–based meta–language used to formalize the structure of a XML document. In the context of WSOA, it is used to specify the data structures exchanged between services.
- **WSDL** The Web Service Description Language [W3C, 2001] is used to describe service interfaces in an implementation–independent way. The WSDL interface of a service describes the different operations provided, grouped by ports, as well as the address to be used to invoke the service. Operation signatures are described as data structures (*i.e.*, XSD artifacts).
- **SOAP** The Simple Object Access Protocol (SOAP) [W3C, 2007] is a protocol used to support the exchange of messages between services. Messages are encoded in a XML dialect, and transported over the network through standard protocols (*e.g.*, HTTP, SMTP).
- **UDDI** The Universal Description Discovery and Integration standard (UDDI) [OASIS, 2004] supports the discovery part of the service model. UDDI registries publish web services interfaces, available for customers. One can query the registry as (i) a *white pages* system (*e.g.*, “give me the FLICKR® service”), (ii) a *yellow pages* system (*e.g.*, “give me a picture repository service”) or finally (iii) as a *green pages* system (*e.g.*, “give me a service accessible over HTTP”).

¹REST web services do not publish explicit interface, and rely instead on self–descriptive messages. As a consequence, it does not intrinsically change the fundamentals of composition mechanisms.

▮ *An interesting thing to notice is that there is no intrinsic support for composition mechanisms in the previously listed standards. In the context of WSOA, composition mechanisms are not standardized!*

Standards are part of the reality, and numerous actors rely on it. Pieces of software are implemented to support standards, e.g., code generators or validation tools. This fact induces the second property required by a platform designed to support composition of services.

$\rightsquigarrow P_2$: **One cannot ignore reality.** De facto standards exist in industry, and composition platforms must be able to reach them.

Based on these standards, the Web Services technological stack provides a strong foundation to define SOA. However, the “simplicity” of the model does not intrinsically support concerns such as transactions, reliability, notification support. To address these issues, several Web Services specifications were defined. These specifications are known under the WS-* acronym. The initial idea of the WS-* specification was to support non-functional concerns at the protocol level. For example, the WS-SECURITY standard [OASIS, 2006b] provides in a 76 page document a specification to support well-known security models (e.g., security tokens) while exchanging SOAP messages. This proliferation of complex standards² leads to three issues: (i) platform vendors are not able to implement all these standards and obviously make commercial choices, (ii) reference implementation of a standard are not mandatory (this point is exposed as one of the reasons which explain the “rise and fall” of CORBA [Henning, 2006]) and finally (iii) the WS-* are not always “necessary” (i.e., the protocol level is not the right place to implement such a concern [Vinoski, 2004]). Actually, these different standards need to be interleaved to support the definition of complex business processes [Curbera et al., 2003].

▮ *Compositions appear now at different levels: (i) the behavior defined as a composition of services and (ii) the policies composed to handle technical concerns.*

A composition platform needs to support the definition of such concerns/policies, and their integration in the business-driven behavior. These two properties are stated as the following:

$\rightsquigarrow P_3$: **Concerns Reification.** Concerns are first class entities and encapsulate knowledge which need to be integrated at the business level. This integration modifies the business-driven behavior.

$\rightsquigarrow P_4$: **Concerns Behavioral Composition.** One needs to manage the way concerns are integrated into the system and its preexisting behavior, to support the definition of complex pieces of software.

2.3 Focus on Behavioral Compositions

We described in the previous sections how the SOA paradigm relies on the “composition of service” notion. This section describes workflow mechanisms used to support the compositions on services, with an emphasis on the Business Process Execution Language (a *de facto* standard for web services composition, used as the validation platform for this contribution).

2.3.1 From Workflows ...

Workflows languages are used to support the abstraction of behavior. The workflow word can refer to two notions: (i) human workflows or (ii) software workflows. In this dissertation, we focus on the second kind of workflows. Software workflow are defined by the SOA4ALL as the following: “The sequence of activities performed in a business that produces a result of observable value to an individual actor” [Soa4All, 2009]. One can find in the literature many different languages used to express workflows [Montagnat et al., 2009], at different levels of abstraction. The

²The (non-exhaustive) list of WS-* standard maintained by Wikipedia exposes 49 standards (complementary or overlapping) promoted by major SOA actors (http://en.wikipedia.org/wiki/List_of_web_service_specifications, 08/23/2010).

more common representation used to reify workflows is based on graphs: steps are represented as nodes, and relations between the steps as arrows between the associated nodes. At the workflow design level, this representation fits business-experts needs, by simplifying a workflow to a simple graph description. At the implementation level, a graph representation allows execution engine to reason on the graphs, and then optimize their execution.

Parallel Composition. Three kinds of parallelism can be identified in a workflow, to optimize their execution. These considerations are critical to obtain efficient compositions. We can identify two different kinds of parallelism at a coarse-grained level: (i) steps which are not dependent between each others can be used in parallel (activity parallelism) and (ii) two steps defined in sequence can be executed in parallel on two separated data sets (data parallelism). Usually, data parallelism are handled by the execution platform [Glatard, 2007]. On the contrary, activity parallelism is defined at the workflow level, where designers can explicitly decide to avoid unexpected wait³.

↪ P_5 : **Activity Parallelism.** Activity parallelism is very important to support the design of efficient workflows. A composition tools must define mechanisms to support it.

Workflow Patterns. The Workflow Patterns Initiative is an academic joint work⁴ which identifies 43 control patterns [van der Aalst et al., 2003, Russell et al., 2006a] used to asses workflow languages and standard. Their study also identify patterns dealing with *resource* handling, *data* management and *exceptions*. The initiative publish in-depth evaluations of several existing languages or standard implementation⁵. The goal of these evaluations is to measure the expressiveness of these languages and standards. The considerable number of patterns raises an expressiveness issue in the definition of compositions. A language cannot implement naturally all of these concepts in its syntax (in their study, there is no evaluated tools which provide such a support). It naturally leads to a choice-diversity issue, and the following property:

↪ P_6 : **Syntax Independence.** Composition tools must be abstracted from their concrete syntax and provide abstract mechanisms to support the definition of workflows.

2.3.2 ... To the Business-Process Execution Language (BPEL)

In the context of web services, workflows can be designed as centralized *orchestrations* or peer-to-peer *choreographies* [Peltz, 2003]. Choreography languages are intended to “define from a global viewpoint the common and complementary observable behavior specifically, the information exchanges that occur and the jointly agreed ordering rules that need to be satisfied” [W3C, 2005]. This standard was aborted by the W3C in July 2009, due to its non-adoption by the industrial nor academic community. On the contrary, the Business Process Execution Language (BPEL [OASIS, 2007]) is defined as an industrial standard used to implement orchestrations, under the name of *business processes*:

“[...] WS-BPEL defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners. [...] The WS-BPEL process defines how multiple service interactions with these partners are coordinated to achieve a business goal, as well as the state and the logic necessary for this coordination. [...]” [OASIS, 2007]

The associated specification is implemented and available in open-source execution engines such as the OPEN ESB⁶ one. Industrials promote the standard by the adoption of BPEL-ready service buses [Bolje et al., 2006], and an involvement in its definition⁷. Academic research propose state-

³Experimental performance benchmarks are designed to check this capability [Din et al., 2008]

⁴<http://www.workflowpatterns.com/>, Eindhoven & Queensland Universities of Technology.

⁵<http://www.workflowpatterns.com/evaluations/>

⁶<http://open-esb.dev.java.net/>

⁷The language derives from the WSFL initial release, published by IBM and Microsoft

of-the-art enhancements to this language. For example, formal languages such as petri-nets can be used to allow model-checking of existing orchestrations [Hinz et al., 2005]. In [Pourraz, 2007], the author proposes a π -calculus driven vision of the BPEL, supporting the verification of architectural properties. The semantic web community provides interesting approaches to support the adaptation of business processes based on semantic descriptions [Küster and Kunig-Ries, 2006]. the scientific workflow community also used BPEL processes to enact workflows on computing grids [Emmerich et al., 2005].

- ▮▮▮ According to P_2 (“one cannot ignore reality”) and to ease the reading of this dissertation, we adopt the vocabulary associated with BPEL and consider “composition of services” as “orchestrations of services”, that is, “business processes”.

2.4 Introducing Separation of Concerns

The previous sections identify the context of the work presented here, that is, composition of services expressed as *orchestration*. In this context, *compositions* are defined as the keystone of the SOA paradigm, used to support complex business-oriented processes. However, the SOA paradigm does not provide natural support (or concepts) to tame the design of complex business processes. The Separation of Concerns (SOC) paradigm [Hürsch et al., 1995] was defined to tackle this issue. The underlying ideas came from E. W. Dijkstra:

“ We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained –on the contrary!– by tackling these various aspects simultaneously. It is what I sometimes have called «**the separation of concerns**», which, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts, that I know of. This is what I mean by «focusing one’s attention upon some aspect»: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously. ”

[Dijkstra, 1974]

Based on this idea, several approaches and programming paradigm emerge to consider programs as the composition of elementary concerns: (i) *subject-oriented* programming which focuses on the identification of composable subject in a program [Harrison and Ossher, 1993], (ii) *role-oriented* programming which considers entities as sharing several roles across their life-cycle [Becht et al., 1999] and (iii) *aspect-oriented* programming [Kiczales et al., 1997] which focuses on the handling of cross-cutting concerns. Another cutting edge research domain which is part of the SOC paradigm is the research made on the Software Product Lines paradigm (SPL, [Clemens and Northtop, 2001]) and Feature Oriented Programming (FOP, [Prehofer, 1997]).

- ▮▮▮ Following the SOC paradigm, concerns are composed to build the final application. One can notice a new shift in the definition of composition: programs are no longer considered as the composition in itself, but as the unit of composition used to build more complex pieces of software.

SOA relies on business-driven processes, written by business experts [Stein et al., 2008]. Considering each concerns as a domain of expertise (e.g., security, persistence, encryption, P_3), each expert can work on its own concern separately. As a consequence, a composition platform needs to take care of concurrent and potentially disjoint work when the concerns will be integrated (P_4).

- ~ Π_1 : **Concurrent experts support.** In the context of SOA, separation of concerns leads to the intervention of several experts, working concurrently on the same system.

2.4.1 Aspect-Oriented Programming

The AOP paradigm is initially defined as the following:

“AOP works by allowing programmers to first express each of a system’s aspects of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using a tool called an Aspect Weaver™. [...] Using AOP, a system is decomposed into its basic functionality and the cross-cutting aspects. (The basic functionality is sometimes called the primary aspect.) The basic functionality and each of the aspects are captured using appropriate special-purpose languages.” [Kiczales et al., 1997]

The key ideas used to support the SOC paradigm are (i) the encapsulation of cross-cutting concerns in separated *aspects* and (ii) the usage of a “weaver” program to integrate aspects into a legacy application. A strong difference is made between the base program (i.e., the “business” code) and the cross-cutting concerns implemented as aspects (e.g., authentication, security, log). We use in this dissertation the ASPECTJ [Kiczales et al., 2001] platform as a reference implementation and vocabulary.

Aspects. An aspect defines (i) an implementation of a cross-cutting concern as plain code and (ii) indications for the weaver to know where the concern should be integrated in the base programs. The AOP vocabulary [ERCIM, 2010] defines *joinpoints* (from subject-oriented programming) as the set of reachable locations in a base program (e.g., a method call) and *pointcuts* as the concrete subset of joinpoints targeted by the aspect (e.g., “the call of the method *m*”). According to Aksit, “The crosscutting concerns of a design represent crucial abstractions. It is therefore important to represent these concerns explicitly, encapsulate their implementation details, and provide operations to extend them. AOP languages make crosscutting concerns more understandable and manageable.” [Elrad et al., 2001]. Consequently, the definition of AOP focuses on cross-cutting concerns capture to tame the complexity of designing large pieces of software. At the execution level, the aspects are executed sequentially in the most common approaches. Dedicated AOP platform must be used to support aspect concurrency [Douence et al., 2006].

In the SOA context, cross-cutting concerns are usually handled by WS-* policies, such as WS-SECURITY. At the implementation level, AOP can be used to support their integration in the web service platform [Verheecke et al., 2003]. However, several cross-concerns such as data-persistence or fault tolerance must be handled at the orchestration level instead of the protocol one [Karastoyanova and Leymann, 2009].

➡ SOC composition mechanisms must be defined to support the integration of concerns in base orchestrations.

Weaver. The role of the weaver is to perform the integration of the aspects into the base program [Masuhara et al., 2003]. A weaver consumes as inputs the base program and the associated aspects. The expressed pointcuts are analyzed against the joinpoints (pointcut matching step) to identify where the aspect code (known under the name of “advice”) must be integrated. The output of the weaver is an “advised” program, which include the base programs and the aspects advices. To support dynamic weaving [Popovici et al., 2002], the weaver must be integrated in the targeted language. Finally, the “advised” program may behave incorrectly due to aspect interferences. Several works address this issue [Katz and Katz, 2008, Aksit et al., 2009] by providing mechanisms to detect interferences and mechanisms to support their resolution. Usually, aspects are ordered to solve these interference. However, the identification of the “right” order is known as a very difficult task [Pawlak et al., 2005]. The main algorithm used to compose aspect with a base program is implemented in the weaver entity. However, the way aspects are scheduled at runtime are handled by another algorithm: the aspect scheduler. This fact leads us to the second compositional property:

↔ Π_2 : **Support for multiple composition algorithm.** Composition of multiple concerns induces different mechanisms to be used while performing the integration. These mechanisms need to be supported by different accurate algorithms.

Aspect-Oriented methodology applied to SOA & BPEL. One can find in the literature several research works which implement AOP mechanisms for SOA. For example, the AO4BPEL framework [Charfi and Mezini, 2004] proposes a modification of a BPEL execution engine to support dynamic weaving of aspects. In [Courbis and Finkelstein, 2005], authors propose to use aspect at different levels (semantic analysis, BPEL engine & BPEL processes) to support the introduction of unforeseen features. These approaches handle AOP at the level of the BPEL engine. From the composition point of view, it supports the integration of BPEL advices inside legacy process. We previously explained why a syntactic-driven approach is not sufficient from our point of view. Moreover, these approaches imply to learn at least two languages (the BPEL and the aspect language) to integrate new concerns into existing process. Other approaches use aspects at the operation level, to modify the internal behavior of operations exposed by services [Baligand and Monfort, 2004, Ortiz and Leymann, 2006]. The Service Component Architecture specification [OpenSOA, 2007] can be used to bind component and services in a more global view [Seinturier et al., 2009]. As a consequence, component-based approaches can be adapted to SOA through this platform. In [Barais et al., 2008b], the authors manage the integration, in component assemblies, of new concerns represented as architectural aspects. This work relies on transformation rules describing precisely how to weave an architectural aspect in an assembly. Transformation engine manages the composition of the aspects. The behavioral modifications are then the consequences of architecture changes. In [Pessemier et al., 2008], at component level, the authors express a similar point of view defining aspect components as encapsulation of advice codes.

↪ **Is AOP the expected paradigm?** Based on this description, we can confront⁸ AOP to the properties identified to support the design of complex business processes (P_x). According to its initial definition, the AOP was designed to reify non-functional⁹ concerns as first class entities ($P_3[+]$) and supports their integration ($P_4[+]$). However, the paradigm was not designed to support business-driven expressiveness ($P_1[-]$), and is strongly coupled with the underlying language ($P_6[-]$). SOA standard can be reached iff the underlying language implements associated API ($P_2[\sim]$). The aspect ordering mechanisms does not allow an immediate support for activity parallelism ($P_5[\sim]$) (see CHAP. 7).

2.4.2 Feature-Oriented Programming

Where AOP expresses cross-cutting concerns to be integrated into a base program, Feature-Oriented Programming (FOP) proposes to build an application as the composition of coarse-grained features. These two approaches are designed to achieve different goals, and are complementary [Apel and Batory, 2006]. We consider here the FOP implementation provided by the AHEAD tool suite [Batory et al., 2004]. These tool suite is a direct descendant of the GENVOCA tool [Batory and O'Malley, 1992]. AHEAD is used to support a *step-wise* development approach, defined as the following:

“Stepwise refinement is a powerful paradigm for developing a complex program from a simple program by incrementally adding details [Dijkstra, 1997]. The program increments that we consider in this paper are feature refinements modules that encapsulate individual features where a feature is a product characteristic that is used in distinguishing programs within a family of related programs (e.g., a product line) [Griss, 2000].” [Batory et al., 2004]

An interesting specificity of the FOP paradigm is its natural representation through plain mathematical notations [Batory, 2008]. The composition operator is denoted as \bullet , and corresponds to the usual function composition $f \bullet g(x) = f(g(x))$. The $f \bullet g$ notation means “add the feature f to the program g ”. According to this functional representation, a system is defined as

⁸ $P_x[+]$ means that this property is satisfied by the paradigm, and on the contrary $P_x[-]$ identifies a lack. The $P_x[\sim]$ notation denotes an indirect support.

⁹ e.g., performance, security [Chung and do Prado Leite, 2009]

a composition of features. This algebraic representation allows the usage of equation optimizers to rewrite the compositions in an efficient way [Liu and Batory, 2004]. This property is identified as the following:

$\rightsquigarrow \Pi_3$: **Compositions as first class entities.** Compositions must be reified as dedicated entities, and then automatically analyzed to enhance their efficiency / correctness.

According to the FOP paradigm, features are sequentially composed (layered) to obtain the final system. Without any extra-information, it is clear that $f \bullet g$ and $g \bullet f$ produce different systems: $f \bullet g(x) = f(g(x)) \neq g(f(x)) = g \bullet f(x)$. The order of composition is consequently important. Several methods can be used to support the definition of this order. It is possible to reify the interaction of a feature and another one [Liu et al., 2005] using mathematical derivative function. Another lead is to use commuting diagrams [Kim et al., 2008] to explore the different composition orders. The way features are composed together can be constrained through the usage of *design constraint rules* [Batory and Geraci, 1997], expressed as attributed grammars [McAllester, 1994]. A “valid” composition is consequently identified as a word recognized by the design constraint grammar. This approach allows the user to identify conflicting features upstream. Software product lines can be checked to identify inherent conflicts in some extents [Kästner et al., 2009], and ensure safe composition. These detections can be done directly on the product line [Apel et al., 2010], or on the final product [Perrouin et al., 2008]. The identification of conflict ensures “safe-composition”, and leads us to the following property:

$\rightsquigarrow \Pi_4$: **Conflict detection mechanisms.** Conflicting entities need to be properly identified by the composition platform.

At the implementation level, the AHEAD tool suite provides command-line tools to support the composition of features [Batory, 2006]. One can notice the definition of several composition algorithms (e.g., **mixin** & **jampack**), which enforces the Π_2 property: different goals need different composition algorithms. The tools compose class-based structure, and behavior expressed as state-machine.

Feature-Oriented methodology applied to SOA. FOP is used to support the synthesis of SOA. Based on a set of user-given requirements, the associated services are synthesized in a generative approach [Batory, 2007]. This approach leads to several issues & challenges, described in [Apel et al., 2008]. The most important one is the “black-box” assumptions on services: code-composition techniques cannot be directly used to pervasively modify the internal code of a service, as such a service is considered to be a black-box, independent from any implementation.

\rightsquigarrow **Is FOP the expected paradigm?** Features are definitively designed to fulfill a business-oriented vision of software design ($P_1[+]$). Concerns are reified as identified and independent artifacts ($P_3[+]$). Like AOP, the SOA standards can be reached with the associated implementation step ($P_2[\sim]$). Behavioral composition is driven by state machines, which naturally support parallel composition ($P_5[+]$) but compel the designer to use such a formalism ($P_4[\sim]$). The FOP paradigm is obviously syntax-independent, as one can design features involving different languages (e.g., Java, XML description and Makefile compilation instruction). However, each language should be operated by the associated composition tool in order to be supported by the approach ($P_6[\sim]$).

2.5 Summary & Contribution Choices

Summary. In SEC. 2.2 and SEC. 2.3, we identified properties required by a platform in order to support the design of complex business processes. Then, we confront these properties with two SOC of concerns approaches: AOP and FOP. We summarized the result of these comparisons in TAB. 2.1: there is no immediate support for all of these properties in both AOP and FOP.

SOA Composition Property		AOP	FOP
P_1	Business expressiveness	–	+
P_2	One cannot ignore reality	~	~
P_3	Concern Reification	+	+
P_4	Concerns Behavioral Composition	+	~
P_5	Activity Parallelism	~	+
P_6	Syntax Independence	–	~

Table 2.1: Summary of expected properties and their implementation in AOP & FOP

Towards a SOA–dedicated composition support. We propose an intermediate approach to fulfill this goal. This approach will be designed exclusively for the context of SOA. This domain–specialization allows us to strengthen the previously described approach in order to support the definition of complex business processes.

- P_1 . *Business expressiveness.* From FOP, we keep the business orientation associated to a feature. We do not categorize the concerns as “functional” and “non–functional”. Concerns are expressed with the same concepts than legacy processes, and stay at the same level of expressiveness.
- P_2 . *One cannot ignore reality.* The approach will not diverge from the WSOA standard described in the previous section.
- P_3 . *Concern Reification.* Concerns will be considered as first class entities, like aspect or features.
- P_4 . *Concerns Behavioral Composition.* Concerns will express pieces of behavior that aim to be integrated into preexisting ones. We restrict this work to the composition of behaviors, and consider complementary tools to address the composition of structural data.
- P_5 . *Activity Parallelism.* The approach will naturally support activity parallelism.
- P_6 . *Syntax Independence.* Syntax independence is not reachable if one stays at the language level. As a consequence, we will follow a model–driven approach and reason on models instead of code (see CHAP. 3).

Composition properties expected in the platform. The previously described properties emphasize the SOA domain specialization, at the business process level. We consider now the intrinsic properties needed by the composition platform in itself:

- Π_1 *Concurrent experts support.* A stepwise development will be available when the concern order is known. But the composition mechanisms will be designed to support the concurrent definition of concerns, without explicit ordering. Experts can then work concurrently, and only specify a partial order on their concerns. Ordered concerns are composed like layered features (see CHAP. 7), and unordered concerns are handled in a particular way to provide such a support (see CHAP. 8).
- Π_2 *Support for multiple composition algorithms.* Both AOP and FOP defines different composition algorithms. The approach will include mechanisms to define new composition algorithms (see CHAP. 6).
- Π_3 *Composition as first class entities.* The indication given by the process designers will be used to reason about the given composition, and “optimize” it. We propose a full development method, based on the scheduling of several composition algorithm inside a step (see PART III).
- Π_4 *Conflict detection mechanisms.* The processes obtained after composition will be checked to identify issues generated by the composition (see CHAP. 7). These conflict detection mechanisms will include SOA–dedicated properties.

We summarize in TAB. 14.1 (p. 191) how the contribution described in the next chapter respects this properties. We believe that a concise summary used to conclude this work according to these properties will be more simple to follow than a diffused version where properties are evocated chapter by chapter.

Composing Behavioral Models

«Do not quench your inspiration and your imagination; do not become the slave of your model.»

Vincent van Gogh

Contents

3.1 Introduction	21
3.2 Towards a Model-Driven Approach	22
3.2.1 Models & Meta-Models	22
3.2.2 Immediate Benefits of Model-Driven Engineering	24
3.2.3 Requirements Summary & Leading Choices	24
3.3 Cutting Edge Behavioral Modeling Formalisms	25
3.3.1 Business Abstraction (BPMN)	25
3.3.2 Graphical Syntax (UML Activity Diagrams)	26
3.3.3 Execution Semantics (Protocol Modeling)	28
3.4 Model-Driven Compositions Mechanisms	28
3.4.1 Sequence Diagrams Weaving	28
3.4.2 Conflict Detection	29
3.4.3 Parallel Composition	29
3.4.4 Multi-view modeling	29
3.5 Summary & Contribution Choices	30

3.1 Introduction

In the previous chapter, we identified several properties that support the definition of complex business processes (P_x), following a Separation of concerns (SOC) approach (Π_x). It leads us to the “specification” of a SOA-dedicated composition tools. However, we described the “syntax proliferation” issue in the context of SOA, which is still unanswered. We propose to use a model-driven approach to tackle this issue. The key idea is the definition of a *small* meta-model (see CHAP. 4), preserved from flourishes added by each workflows language. This “kernel” will be used as a solid foundation for the platform described in the previous chapter. The goal of this chapter is to motivate why we define our own meta-model instead of relying on an existing one. We identify requirements (R_x) expected in a model-driven platform to support the design of complex business processes, according to a SOC approach. These requirements are then confronted with the existing formalisms.

The chapter is organized as the following: SEC. 3.2 briefly presents the Model-Driven engineering (MDE) approach, following a pragmatic rationale. We present in SEC. 3.3 formalisms used to support behavioral modeling, and SEC. 3.4 describes several model-driven approaches

which support behavioral composition. Finally, we summarized in SEC. 3.5 the benefits identified in these approaches which are reused in this contribution.

3.2 Towards a Model-Driven Approach

Working at the code level to identify and validate concerns inter-dependencies is known to be a difficult problem (e.g., the ASPECTOPTIMA case study [Kienzle et al., 2009b]). Based on this fact, it seems obvious to raise the level of abstraction, and follow a model-driven approach. We use the following definition of “modeling” as starting line for this study:

“Why do we model?” Modeling is a central part of all the activities that lead up to the deployment of good software. We build models to communicate the desired structure and behavior of our system. We build models to visualize and control the system’s architecture. We build models to better understand the system we are building, often exposing opportunities for simplification and reuse. We build models to manage risk.” [Booch et al., 2005]

In the context of SOC and SOA, the keypoint of the previous definition is “to understand the system”. In our case, the system is composed by several business processes which need to be composed to build a more complex one. Staying at the code level compel us to reason about low-level artefacts (i.e., a syntactical representation of the business process), each with its own specificity (e.g., depending on the used language).

↪ R_1 : **Need for Abstraction.** Reasoning at the model level is mandatory to better understand our system. To understand the composition mechanisms needed to support the design of complex business processes, we will use an abstraction of the business processes core elements (i.e., activities and relations between activities).

In the same book, authors expose four principles (guidelines) used as the foundation of a modeling-based approach. These principles can be seen as guidelines for modeling.

“Principles of Modeling. (i) The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped. (ii) Every model may be expressed at different levels of precision. (iii) The best models are connected to reality. (iv) No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.” [Booch et al., 2005]

Based on these principles, we identify two points of view. First of all, models must be dedicated to solve a dedicated problem, anchored in the reality (principles (i) & (iii)). Secondly, several models are needed to solve different problems, and the approaches must be complementary (principles (ii) & (iv)).

↪ R_2 : **Need for Problem-Specialization.** In our context, the problem is to compose several business processes into a more complex one. We will use in this contribution a modeling approach dedicated to this single purpose.

3.2.1 Models & Meta-Models

There is no “accepted” definition of what is a model. In [Muller et al., 2009], the authors defend “that we are using models, as Molière’s Monsieur Jourdain was using prose, without knowing it”. They identify nine (!) different definitions associated in the literature to the *model* word. We provide here the four more relevant to our topic:

- “Models provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones.” [Brown and Alan, 2004]
- “A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made.” [Kühne, 2006]
- “A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.” [Bézivin and Gerbé, 2001]
- “A model of a system is a description or specification of that system and its environment for some certain purpose.” [Miller and Mukerji, 2003]

The two first definitions enforce R_2 : models support the resolution of problems. The two last definitions raise a related requirement, based on the *purpose* notion.

$\leadsto R_3$: **A purpose, a model.** Models must be defined with for a given purpose. In our case, the purpose is the composition. The modeling approach described in this contribution focuses on this purpose.

Meta-models. A meta-model is defined as the model of models [OMG, 2006]. It contains structural information used to express models *conform to* itself. If we draw a parallel with *grammarware* [Wimmer and Kramler, 2006], a meta-model M is assimilated as a grammar G , and a model m conforms to M is a word w accepted by G . As a consequence, a meta-model contains all the “concepts” needed to express models. In our case, a meta-model dedicated to the representation of business processes will contains concepts such as *activity* and *relation between activities*. However, meta-models should not be considered as *paper-only* artifacts. Powerful tools exist (e.g., the Eclipse Modeling Framework EMF [Merks et al., 2003]) to equip meta-models and generate “things” (e.g., code, documentation) based on their definitions. The Meta-Object Facility (MOF [OMG, 2006]) is defined as a meta-meta-model¹ and provides a reference standard used to define meta-models in both industrial and research tools. Several meta-models were defined to support the design of business processes using EMF as pivot representation. We describe in SEC. 3.3 why these approaches are not sufficient to support the definition of composition mechanisms.

Model transformations. Models (and meta-models) can be represented using “common” formalisms (e.g., XMI [OMG, 2007]). Model transformation engines [Czarnecki and Helsen, 2003] support the implementation of transformation chains between tools (e.g., AGG [Taentzer, 2004], ATL [Jouault and Kurtev, 2006]), based on these formalisms. These approaches are declarative, and rely on pattern-matching mechanisms. An alternative is to use a meta-language such as KERMETA to implement a transformation [Muller et al., 2005]. In a sense, a transformation is “*simply*” a matter of manipulating a model to produce another one. As a consequence, meta-languages dedicated to model manipulation are particularly adapted to support the implementation of model transformation. Independently of the chosen paradigm to implement a transformation, the principles are the same. Let s a system, and m a representation of this system according to a meta-model M (denoted as $m = \mu(s, MS)$). According to R_2 and R_3 , this representation of s follows a given *goal*, supported by M . Considering another meta-model M' , a transformation τ can be defined to transform m into a model $m' = \tau(m)$, conform to M' . For example, a design model (m) can be formally checked in another formalism (m'), thanks to the associated transformation. This transformation approach leads us to the fourth requirement.

$\leadsto R_4$: **Be complementary, and transform if necessary.** Models are transformed to cross the boundary of a given goal, using well-defined transformation languages and formalism, to ensure interoperability.

¹i.e., a set of related concepts used to express meta-models, published as a meta-model itself.

3.2.2 Immediate Benefits of Model-Driven Engineering

The key idea of model-driven development is the central place accorded to models in the software development cycle. Models are then considered as “productive” entities, with associated tools.

“Model Driven Development (MDD) raises the level of abstraction of the development life cycle by shifting its emphasis from code to models, metamodels and model transformations. It views any software artifact produced at any step of the development process as a valuable asset by itself to reuse across different applications and implementation platforms so as to cut the cost of future development efforts.” [Blanc et al., 2005]

The definition of common formalisms supports tools interoperability through model transformations [Sun et al., 2009], as stated in R_4 . Based on this approach, a model-driven approach can benefit from many research and industrial tools, in their respective domain of expertise. We consider here two “domains”, chosen according to their relations with the previously identified properties: (i) coherency (Π_1 & Π_4 , multiple experts working concurrently may generate conflicting models) and (ii) domain-specific languages (P_1 & P_3 , business expressiveness).

Coherency. According to Π_1 , several business experts may interact with the system, concurrently. Several cutting-edge researches focus on the support of collaborative design in model-driven environment. In [Sriplakich et al., 2006], authors propose MODELBUS, a tool dedicated to support collaborative development in model-driven environment. The tool supports mechanisms used to merge concurrent changes on the models, based on structural coherency checks. Conflicts are identified (Π_4), and can be automatically solved in some extents. The immediate successor of MODELBUS is the PRAXIS tool [Blanc et al., 2008, Blanc et al., 2009], which ensures model coherency using an action-based approach. According to this approach, a model is considered as the set of elementary actions used to build it. Based on these descriptions, coherency rules are defined as construction *patterns*, and a logic-based inference engine is used to identify violations. PRAXIS is integrated in a peer-to-peer framework to support real-time collaboration in an efficient way [Mougenot et al., 2009].

Domain Specific language (DSL). DSL [Fowler, 2010] are very important in our context, according to the business-driven property P_1 . A DSL is a language dedicated to a given domain, which intensively uses the essence of the domain in its syntax [Parr, 2009]. Generative programming [Czarnecki et al., 2002] is used to target preexisting languages, which act as execution platforms for the DSL. State-of-the-art research on the relations between meta-models and DSL’ grammars [Muller et al., 2008], proposing to automatically support the transformation from one to the other. Using tools such as SINTAKS [Muller and Hassenforder, 2005] or EMF-TEXT [Heidenreich et al., 2009], one can obtain easily a textual syntax associated to a meta-model. Graphical syntax can be obtained through dedicated framework such as the Graphical Modeling Framework (GMF [Kolovos et al., 2009]).

3.2.3 Requirements Summary & Leading Choices

The previously identified requirements R_x identify the two following facts: (i) a model-driven approach is dedicated to a given purpose and (ii) completes others approaches, dedicated to other purpose. In the previous chapter, we identify our purpose as the following:

“To support the design of complex business processes following a compositional approach”

There is no approach described in the literature which fulfill this specific goal. However, one can found cutting-edge research which deals with (i) business processes modeling and (ii) model composition. The remainder of this chapter consists of a description of these approaches, adopting a complementary point of view. We try to identify reusable mechanisms we can use in existing approaches to tackle our own specific problem.

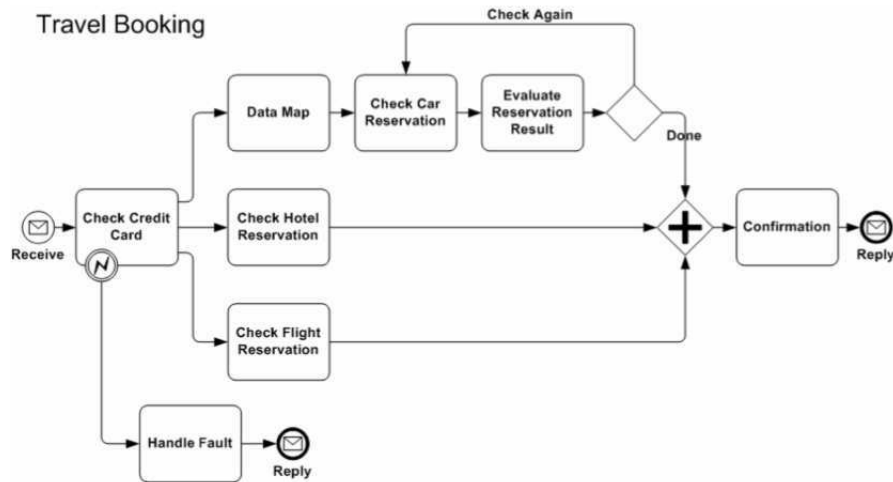


Figure 3.1: BPMN formalism example: the travel booking process [White, 2005]

3.3 Cutting Edge Behavioral Modeling Formalisms

The goal of this section is to describe cutting edge research used to model system behavior. Our goal is to identify complementary formalisms and reuse some of their choice in our own contribution. We organize these formalisms according to the three following criteria: (i) business abstraction, (ii) graphical syntax and (iii) execution semantics. Paragraphs identified by a (†) symbol are based on an article currently under review written with the authors of the associated methods.

3.3.1 Business Abstraction (BPMN)

The Business Process Modeling Language (BPMN [OMG, 2006a]) is defined to support the modeling of business processes. As defined specifically for this purpose, the language fits the abstraction needed to support such a design. It defines several **Elements** used to model business processes. The notation defines 11 concepts, sub-categorized according to their kind. The language rely on **Scope**, represented as *swimlanes* in the notation. Each **Lane** represents a given subset of activities, grouped by role, in a given organization (depicted as a **Pool**). An **Activity** can be instantiated as (i) a **Task** (i.e., black box), (ii) a **Subprocess** (i.e., white box) or (iii) a **Transaction** (i.e., a sub-process which needs to be compensated in case of failure). From the relation point of view, three kinds of **ObjectConnection** can be defined to schedule the activity set: (i) **SequenceFlow** to schedule activities inside the same **Pool**, (ii) **MessageFlow** to represent cross-organizational relations (i.e., an arrow crossing the boundary of a **Pool**) and (iii) **Association** to bind an activity with a **Data** resource. Despite its intrinsic complexity, the language is clearly identified as suitable to support the design of business processes by business experts [Wohed et al., 2006]. We represent in FIG. 3.1 a travel agency process using this formalism.

The specification came with an informal mapping to the BPEL execution language. Such a dichotomy between modeling and execution is clearly identified in [Kloppmann et al., 2009]. However, several conceptual mismatches were identified between the BPMN modeling elements and the BPEL syntax [Recker and Mendling, 2006]. As a consequence, there is no “accepted” transformation from BPMN models to BPEL execution code. One can found in the literature several transformations, based on different interpretations of the BPMN standard [Ouyang et al., 2006, Pant, 2008].

↪ **Key Points assimilated in our approach.** *From the BPMN, we will keep the business-driven expressiveness. However, we will propose a reduced number of concepts, and then limit potential misinterpretations.*

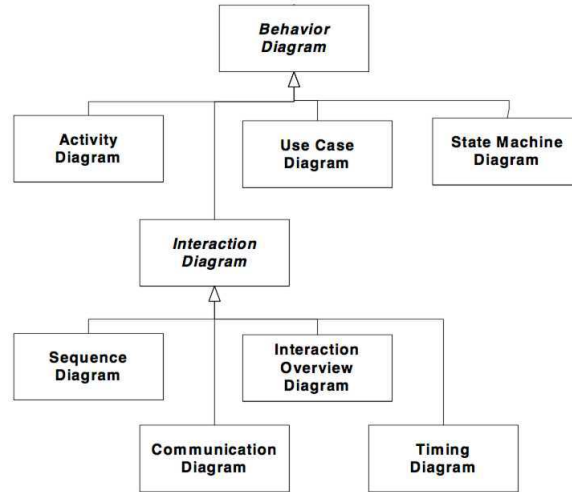


Figure 3.2: Taxonomy of UML behavioral diagrams [OMG, 2005]

3.3.2 Graphical Syntax (UML Activity Diagrams)

The UML super-structure [OMG, 2005] identifies two classes of diagrams: (i) structural and (ii) behavioral. We represent in FIG. 3.2 the class hierarchy associated to behavioral diagrams. The activity diagram formalism is the only UML notation which was clearly demonstrated as suitable to represent business processes [Russell et al., 2006b]. As a consequence, we based our analysis on this notation. An example of activity diagram is depicted in FIG. 3.3.

Contrarily to the BPMN, the activity diagram notation is not dedicated to business process modeling. However, the notation represents “activities” as boxes, and “relations between activities” as arrows. This simple notation supports the definition of arbitrarily complex behavior. However, we identify two drawbacks in this notation according to our goal: (i) the syntactic level and (ii) the lack of an accepted semantics associated to the notation.

- **Syntactic approach.** The activity diagrams formalism can be used as a workflow specification language [Dumas and ter Hofstede, 2001]. The formalism can also be used to specify data-flows, which can be verified and validated with associated tools [Störle, 2004]. It immediately raises an abstraction issue: the formalism defines a syntax (another one ...), but does not reach the domain-specific abstraction expected by R_1 . According to a study published in [France et al., 2006], plain UML is not designed to support a model-driven development approach “as is”, and need to be operated at a higher level of abstraction.
- **Execution Semantics.** One can find in the literature *different* execution semantics associated to the activity diagram notation: there is no “accepted” semantics in the modeling community. Formal methods such as Abstract State Machine [Börger et al., 2010] or Petri-nets [Eshuis and Wieringa, 2001] can be used to support the definition of these semantics. In [Eshuis, 2002], the author defined two completely different semantics associated to activity diagrams, allowing the usage of these diagrams at both implementation and requirement levels. The Executable UML was designed to provide an execution semantics associated to UML diagrams [Mellor and Balcer, 2002]. Unfortunately, they do not consider activity diagrams in their formalisms.

↪ **Key Points assimilated in our approach.** *From activity diagrams, we will kept the notation simplicity, considering activities as boxes and relations between activities as arrows. This choice eases the reading of a diagram, which is based on a well-known representation.*

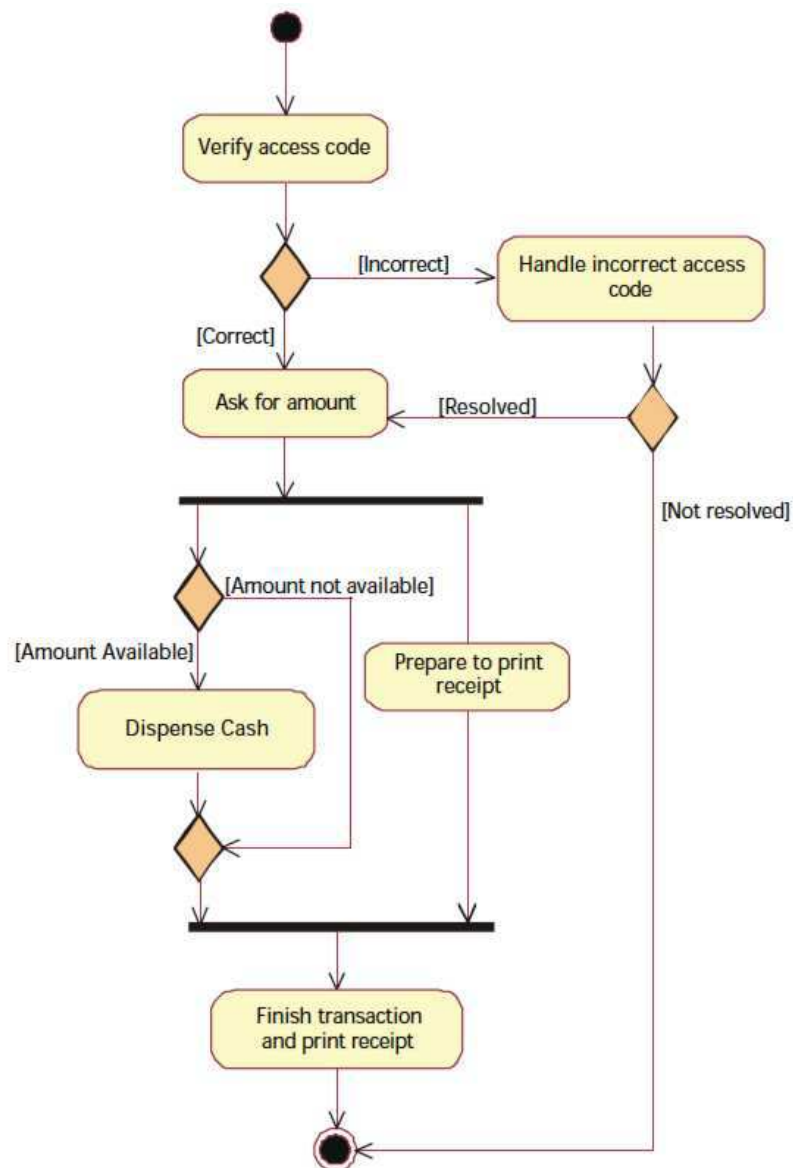


Figure 3.3: Example of an ATM activity diagrams [Ericsonn, 2000]

3.3.3 Execution Semantics (Protocol Modeling)

(†) A Protocol Model [McNeile and Simons, 2006] of a system is a set of protocol machines (PMs) that are composed to create the behavior of the system. The specification of a PM includes its *local storage* and the *alphabet* of *event types*. The local storage is represented as a set of its attributes. Each event type is specified by meta-data. An event instance comes from the environment and is atomic. Instances of PMs are created with happening of events. PM instances can be included in other PMs. A PM instance behaves so that it either accepts or refuses an event from its alphabet, depending on its state and the state of other included machines. Events that are not in its alphabet are ignored. To evaluate state, a machine may read, *but not alter*, the local storage of the included machines. In [Roubtsova et al., 2009], authors propose a complete case study where protocol modeling is used to support the modeling of a complex system. The main focus of the PM approach is specification of the interaction between the system and the environment [McNeile and Roubtsova, 2010]. The semantics of event acceptance is extended with the semantic of event refusal. This allows for modeling of the situation when events occur and the system cannot accept them. The behaviour of the system in a PM model is a set of traces of events accepted by the system.

PMs are good candidates to prototype the interaction of a system and its environment, but do not fit the right level of abstraction associated to business process design in our context. Business processes need to be expressed in business terms, and PMs rely on “machines”, “events” and “alphabet”.

↪ **Key Points assimilated in our approach.** *From PMs, we keep the idea of a strong execution semantics associated to our models. This execution semantics must be simple to understand and verify, and supported by a formal model (e.g., first-order logic).*

3.4 Model-Driven Compositions Mechanisms

In this section, we describe several cutting-edge composition methods applied on models, to support behavioral composition. For each described approach, we identify the key-points which can be reused in our own approach to support the design of complex business processes.

3.4.1 Sequence Diagrams Weaving

In [Klein et al., 2007], the authors propose a method dedicated to support the weaving of aspects using sequence diagrams formalism. Behavioral aspects are defined as sequence diagrams too. The approach is fully described in [Klein, 2006], and integrated into a multi-view modeling approach (RAM, [Kienzle et al., 2009a]). Aspect pointcuts are defined as events captured by the aspect and expressed in the base sequence diagram. The key idea of the approach is to support the matching of message sequences using a *partOf* semantic: an pointcut capturing the sequence of messages $m_1 \rightarrow m_2$ will match a longer sequence such as $m_1 \rightarrow \mu \rightarrow m_2$. This mechanisms supports the introduction of multiple aspects on the same point, as the events added by an aspect will not interfere with the others. According to its weaving mechanisms [Klein et al., 2006], the approach supports the usage of multiple events as target, where usual AOP techniques capture a single one. They propose several matching strategies to catch messages sequence (*i.e.*, *enclosed part*, *general part* and *safe part*), and let the user choose the one which fits its goal. However, the approach is dedicated to sequence diagrams (and the associated structural concerns described in class diagrams) which cannot be used “as is” to represent business processes. Finally, the composition operator defined in the approach rely on activity ordering, and does not support naturally the activity parallelism identified in P_5 .

↪ **Key Points assimilated in our approach.** *From Klein’s method, we keep three key points (i) the usage of the same formalism to represent both aspect and base program, (ii) the support of multiple aspects weaving on the same point and (iii) the use of multiple activities as potential target of an aspect.*

3.4.2 Conflict Detection

The MATA approach [Whittle et al., 2009] supports the weaving of models aspects using a graph-based approach. This approach supports powerful conflict detection mechanisms, used to support the “safe” composition of models [Mussbacher et al., 2009]. The underlying formal model associated to this detection is based on critical pair analysis [Heckel et al., 2002]. Initially defined for term rewriting system and then generalized to graph rewriting systems, critical pairs formalize the idea of a minimal example of a potentially conflicting situation. This notion support the development of rule-based system, identifying conflicting situations such as “the rule r will delete an element matched by the rule r' ” or “the rule r generate a structure which is prohibited according to the existing preconditions”. These mechanisms were demonstrated as relevant to identify composition conflicts in the software product line domain [Jayaraman et al., 2007]. However, these kind of conflicts are driven by the syntax of the graph transformation language, and must be operated to reach the business level expressiveness.

↪ **Key Points assimilated in our approach.** *From MATA, we keep need for conflict detection mechanism to support the “safe”² composition of business processes.*

3.4.3 Parallel Composition

(†) The complete system is composed using CSP parallel composition [Hoare, 1978]. It serves as a natural weaving algorithm for aspects. The alphabet of the composed machine is the union of the alphabets of the constituent machines; and the local storage of the composed machine is the union of the local storages of the constituent machines. When presented with an event the composed machine will accept it if and only if all its constituent machines, that have this event in their alphabet, accept it. If at least one of such machines refuses the event it is refused by the composed machine. The concept of event refusal is critical to implement CSP composition for composition of protocol machines [McNeile and Roubtsova, 2008]. This composition guarantees a property of local reasoning on behaviours of PMs about the behavior of the complete system. However, all the reasoning are specialized for each given instance of a PM, and it is not possible to express “meta-properties”.

↪ **Key Points assimilated in our approach.** *From PMs compositions, we keep the intrinsic support of parallel composition (P_A). Our contribution will not introduce order if not explicitly needed.*

3.4.4 Multi-view modeling

Multi-view modeling approaches are used to support the design of complex systems, based on different points of views. In RAM (Reusable Aspect Models [Klein and Kienzle, 2007]), one can use both class diagrams and sequence diagrams to design a system. The approach supports the concurrent weaving of these two kinds of artifacts, and maintains consistency. The class diagrams are merged according to [Reddy et al., 2006], and sequence diagrams according to Klein’s method [Kienzle et al., 2009a]. This approach focuses on the definition of low-level aspects. The Theme/Uml approach [Carton et al., 2009] is another interesting multi-view modeling approach [Barais et al., 2008a], very similar to RAM. However, the Theme/Uml approach does not support model composition mechanisms.

↪ **Key Points assimilated in our approach.** *The contribution of this thesis is dedicated to the composition of business processes, that is, behavior-driven entities. As a consequence, the approach will be integrated into a multi-view modeling approach if one needs to combine behavior and structure for example.*

²In our context, “safe” means without conflicts.

3.5 Summary & Contribution Choices

There is no model-driven approach described in the literature which tackles the same problem than the one described in this dissertation. However, cutting-edge researches dealing with behavior modeling and model composition provide a powerful source of inspiration to the definition of a new approach. These “inspiration points” were identified in this chapter.

Based on these points, the contribution of this thesis is stated as the following:

- From the behavior representation point of view:
 - A formal meta-model will be defined to specifically support the design of business processes, using a business expressiveness instead of a syntactic approach.
 - A graphical notation inspired by well-known formalism will be used to support the understanding of the meta-model
 - A formal execution semantic will be associated to the meta-model, to support the definition of composition algorithms
- From the composition point of view:
 - The approach will support the weaving of new concerns on *blocks* of activities
 - The activity parallelism will be part of the composition process, and no order will be introduced by the composition mechanisms (although explicitly needed).
 - The approach will focus on behavioral compositions, and will be defined as complementary with structural approaches.
 - Conflict detection mechanisms will be defined to support the validation of the composition results.

State-of-the-art. In this part, we describe research works relevant to the topic of this contribution. Our goal was to clearly identified where the contribution of this thesis is located according to cutting-edge researches on the same domain.

Towards the specification of a model-driven framework. Based on this study of the literature, we identified the key points of our contribution, that is, the support of complex business processes design through a compositional approach. The remainder of this dissertation focus on the description of this work. We provide here a coarse grained overview of the contribution.

We will adopt a model-driven approach, and then propose a meta-model used to support the design of business processes. The meta-model is dedicated to support the composition of business processes, and consequently embed concepts associated to this notion. Even if this meta-model is by essence different from the ones existing in the literature, we will use a graphical notation close to the one defined by UML activity diagrams, to support the intuitive understanding of the models. Finally, we will equip this meta-model with a strong execution semantics. (PART II)

Based on this meta-model, we identified as critical that different composition mechanisms need to be supported, to fulfill different goals. We will provide four different composition mechanisms (PART III), used (i) to assess the meta-model described in the previous paragraph and (ii) to support process designers whilst building complex processes. We will validate these algorithms on two large case studies, based on “real-world” systems (PART IV).

Remarks about implementation. All the work presented in this dissertation is implemented, and available for download on the project website^a. The implementation of the framework is used to support the realized case studies, which contains thousands of activities and relations between activities (available for download too). We made the choice to not present the implementation in this document, considering it as the technical fact of the contribution. However, we provide in APP. A and APP. B an overview of the underlying implementation, at a coarse grained level. The interested reader should refer to the website to retrieve the latest information about the implementation.

^a<http://www.adore-design.org>

Part II

The ADORE Kernel

***Notice:** This part defines the thesis foundations. The pressed reader may focus on CHAP. 5, which describes the graphical notation associated to the formal model, and then gives an intuitive idea of the underlying foundations. These foundations are formally described in CHAP. 4. Finally, CHAP. 6 describes how the ADORE models are manipulated to support the description of composition algorithm.*

The ADORE Meta-Model

«I have deep faith that the principle of the universe will be beautiful and simple.»

Albert Einstein

Contents

4.1 Introduction	35
4.2 Coarse-Grained Description	36
4.3 ADORE: the Formal Model	39
4.3.1 Variables (\mathcal{V}) & Constants (\mathcal{C})	39
4.3.2 Activities (\mathcal{A})	40
4.3.3 Relations (\mathcal{R})	41
4.3.4 Business Processes (\mathcal{P})	43
4.3.5 Universe (\mathcal{U})	45
4.3.6 Iteration Policies (\mathcal{I}) & Blocks	46
4.4 Properties	47
4.4.1 Variables	47
4.4.2 Activities	47
4.4.3 Relations	48
4.4.4 Business Process	48
4.4.5 Iteration Policies & Block	49

4.1 Introduction

As described in the previous chapters, a considerable number of technologies were defined to support the composition of services paradigm. According to the MOF, “A *model is a simplification of a system, with a certain purpose*” [OMG, 2006]. Considering our purpose to be the support of orchestration evolution, we describe here the ADORE meta-model, defined for this specific goal.

The ADORE activity meta-model is directly inspired by the BPEL language grammar expressiveness. The rationale of the BPEL is defined as the following:

“The Business Process Execution Language (BPEL) is a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners.” [OASIS, 2007]

The BPEL defines nine different kinds of atomic activities (*e.g.*, service invocation, message reception and response) and seven composite activities (*e.g.*, sequence, flow, if/then/else), plus additional mechanisms such as transactions and message persistence. We decide to extract from

the BPEL all concepts *necessary* “for describing the behavior of a business process”. From the activity expressiveness point of view, ADORE can be seen as a simplification of the BPEL concepts, as explained in the following section. The major shift between the BPEL and ADORE relies in the *links* between activities, used to schedule the process content. We enhance the expressiveness of *links* between BPEL activities (called *relations* in ADORE) to support orchestration evolution through our composition mechanisms.

In this chapter, we describe in SEC. 4.2 the ADORE meta-model following a *top-down* approach (using the ECLIPSE EMF class diagram formalism [Merks et al., 2003]). Then, we define formally each concept used in ADORE in SEC. 4.3, using first-order logic [Rosen, 2000]. We finally express in SEC. 4.4 a set of properties valid over ADORE concepts. These formal concepts will be used in next chapters to support the description of composition algorithms.

4.2 Coarse-Grained Description

The ADORE meta-model defines an **Universe** as its root. This concept contains all the business processes available for the composition algorithms. We define a **BusinessProcess** as a set of variables, a set of activities and a set of relations reifying a partial order between the activities. A business process can define two different kinds of artifacts: (i) as **Orchestration** (which realizes the behavior associated to an **operation** of **service**) or (ii) a **Fragment** (defining an incomplete behavior which aims to be integrated into another process). The associated graphical representation is depicted in FIG. 4.1.

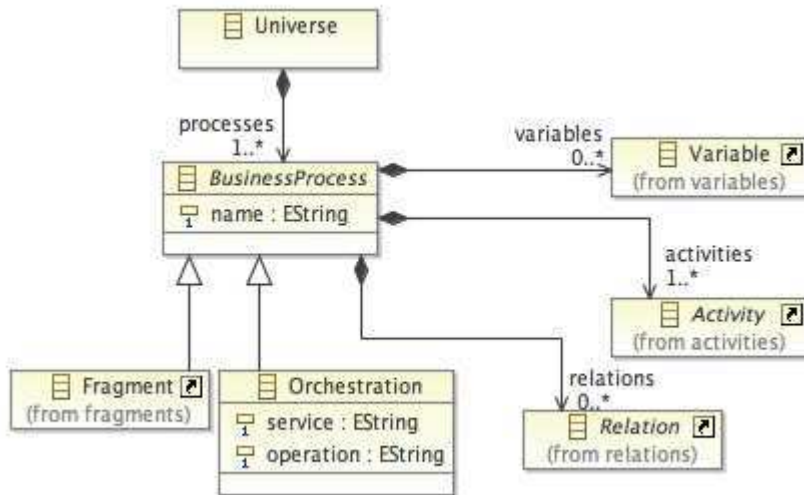


Figure 4.1: Global overview of the ADORE meta-model

The **Activity** concept represents an elementary task realized by a given business process. An activity uses a set of **inputs** variables, and assigns its result in a set of **outputs** variables. The different types of activities that can be defined in ADORE (see FIG. 4.2) include (i) service invocation (denoted by **Invoke**), (ii) variable assignment (**Assign**), (iii) fault reporting (**Throw**), (iv) message reception (**Receive**), (v) response sending (**Reply**), and (vi) the null activity, which is used for synchronization purpose (**Nop**). Consequently, the ADORE meta-model contains all the atomic activities defined in the BPEL normative document except the wait (stopwatch activity) and the rethrow¹.

ADORE defines four different types of relationships between activities (see FIG. 4.3). All relations are expressed using binary operators. ADORE can define simple wait between activities through the **WaitFor** operator. Using **WeakWait**, one can reify predecessors’ disjunc-

¹In ADORE, a rethrow can be performed by *catching* a fault with an **OnFailure** relation and then use a **Throw** activity to re-throw the fault.

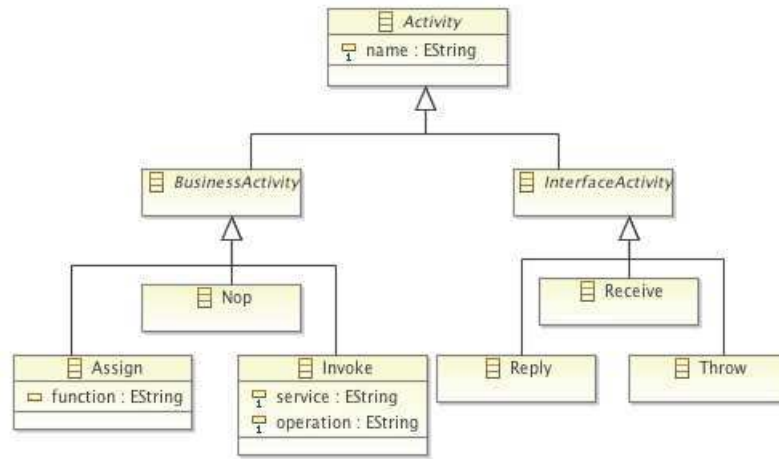


Figure 4.2: ADORE class hierarchy reifying activities kinds

tion [Gregory, 1997], to allow non-determinism in the order of events. Conditional branches in a process control-flow are represented using **Guard** relations. Finally, a fault catching mechanism is defined through the **OnFailure** relations. As the ADORE meta-model does not define composite activities, BPEL composite constructions are reified using the different relations available in the meta-model. A sequence of activities is defined by a **waitFor** relation; If/then/else flows are modeled using **Guard** relations.

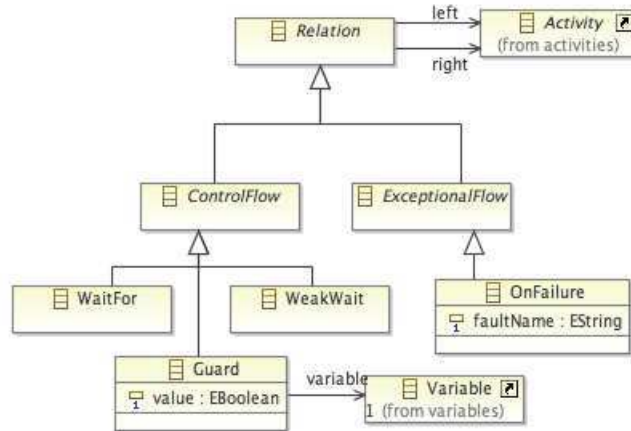
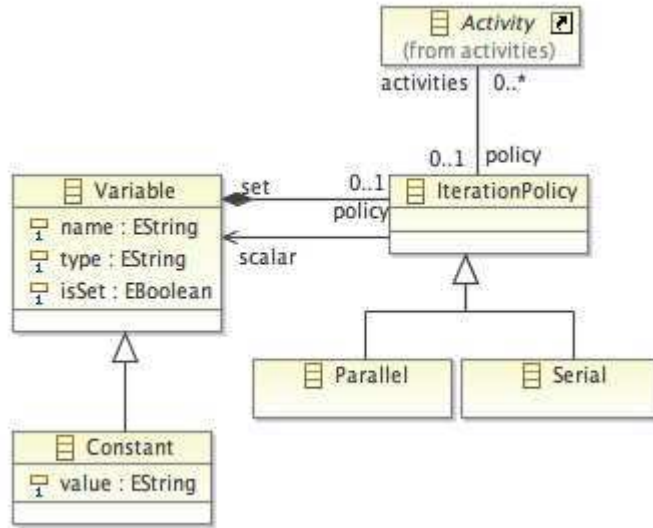


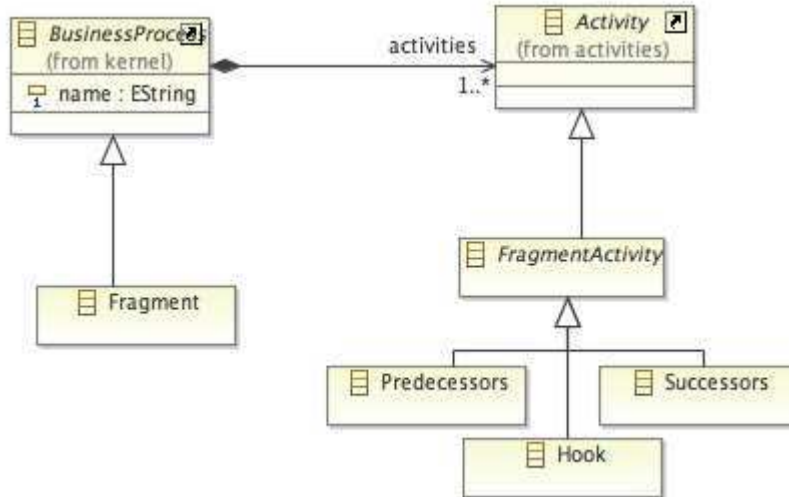
Figure 4.3: ADORE class hierarchy reifying relations kinds

We represent data exchanged between activities using the **Variable** concept (see FIG. 4.4). Additionally to its **name** and **type**, a **Variable** also holds a boolean attribute **isSet**, determining if the entity is representing a set or a simple scalar. **Constants** are defined as extensions of simple variables, and contain an immutable **value**. Contrarily to the BPEL, we do not define control-loops (e.g., for, while) in ADORE. We represent iteration in ADORE using a “for each” mechanism (which can be executed concurrently or in sequence). An **Activity** may be bound to a **Variable** through an **IterationPolicy**. As a consequence, the activity will be executed “for each” data defined in the associated set.

The ADORE purpose is to support the evolution of orchestrations. To perform such a goal, we extend the previously defined meta-model to represent *incomplete* processes. Such processes

Figure 4.4: ADORE concepts associated to **Variables** and **Constants**

(called **Fragment**, FIG. 4.5) are not suitable for direct execution: they aim to be integrated into existing **Orchestrations**. As a consequence, they hold three kinds of *special* activities, dedicated to this goal: (i) a **Hook** which represents the point where the fragment will be integrated, (ii) a **Predecessors** to represent hook' predecessors in the targeted orchestration and finally (iii) a **Successors** to represent hook' successors.

Figure 4.5: ADORE extension to integrate **Fragment** concept

Relations to AOP/AOM. According to the ERCIM working group on software evolution, *aspect-oriented* approaches rely at a syntactic level on four elementary notions [ERCIM, 2010], identified as: (i) *joinpoints*, (ii) *pointcuts* (iii) *advice* and finally (iv) *aspects*. *Joinpoints* represent the set of well-defined places in the program where additional behavior can be added. In the ADORE context, we use activities to reify this notion. *Pointcuts* are usually defined as a set of joinpoints. In ADORE, one can identify sets of activities as pointcuts using explicit declarations (e.g., use

$\{act_3, act_4\}$ activities as pointcuts) or computed declarations (e.g., all activities calling the service *srv*). *Advice* describes the additional business logic to be added in the initial system. We reify *advices* as *fragment*. Finally, *aspects* are defined as a set of pointcuts and advices. There is no explicit *aspect* notion in ADORE. This is done according to the usage of composition algorithms (see PART III).

4.3 ADORE: the Formal Model

In this section, we formally define each ADORE concepts using the may-sorted first-order logical formalism. The goal of this work is to build a solid basis used to support both execution semantic and composition algorithms definition in the following sections. The usage of first-order logic definitions allows us to easily map this formal model into an executable implementation, using the PROLOG language. However, we consider these implementation details as technical (the ADORE implementation is available on the project website², defined as 8,000 lines of PROLOG and 3,000 line of Java wrapper code) and base the remaining of this document on this formalism. A brief description of the implementation is available in APP. A and APP. B.

4.3.1 Variables (\mathcal{V}) & Constants (\mathcal{C})

A variable $v \in \mathcal{V}$ is defined as a *name*, a *type* and a boolean value *isSet?* used to distinguish scalars from data-sets. Both *name* and *type* are defined as ground terms, i.e., terms without any variables.

$$v = (name, type, isSet?) \in (GroundTerm \times GroundTerm \times \mathbb{B}) = \mathcal{V}$$

PICWEB Illustration. We consider here the `getPictures` process defined in the PICWEB requirements. The process receives a *tag* from the user and build a set of pictures URL *picture** as an output.

$$\begin{aligned} tag &= (tag, string, false) \in \mathcal{V} \\ picture^* &= (picture^*, url, true) \in \mathcal{V} \end{aligned}$$

To access *variable* content, we define a set of eponymous functions:

$$\begin{aligned} \forall v = (n, t, i) &\in \mathcal{V} \\ name(v) &= n \in GroundTerm \\ type(v) &= t \in GroundTerm \\ isSet?(v) &= i \in \mathbb{B} \end{aligned}$$

Constants. Constants are defined as variables with an immutable initial value. This concept is reified using a dedicated boolean function *isConstant?*. One can access constant internal value through a dedicated function *value*.

- $isConstant? : \mathcal{V} \rightarrow \mathbb{B}$
- $value : \mathcal{V} \rightarrow GroundTerm$

We denote as $c \in \mathcal{C}$ the fact that *c* is a constant.

$$c \in \mathcal{C} \equiv c \in \mathcal{V} \wedge isConstant?(c)$$

PICWEB Illustration. The `getPictures` process must retrieve a key associated to the FLICKR service before being able to call it. This key is stored in a key repository, under the name *flickr*. We use a *service* constant to define such data in the process.

$$service = (service, string, false) \in \mathcal{C}, value(service) = 'flickr'$$

²<http://www.adore-design.org>

4.3.2 Activities (\mathcal{A})

An activity $a \in \mathcal{A}$ is defined as a *name*, a *kind* $\in Kinds$ (to represent the inheritance graph depicted in FIG. 4.2, see next paragraph), a set of input variables *inputs* and a set of output variables *outputs*.

$$\begin{aligned}\mathcal{A} &= (GroundTerm \times Kinds \times \mathcal{V}^* \times \mathcal{V}^*) \\ a &= (name, kind, inputs, outputs) \in \mathcal{A}\end{aligned}$$

To access *activity* content, we define a set of eponymous functions. We also define a *vars* function to access to the used variable without any consideration on their directions.

$$\begin{aligned}\forall a = (n, k, in^*, out^*) &\in \mathcal{A} \\ name(a) &= n \in GroundTerm \\ kind(a) &= k \in ClosedTerm \\ inputs(a) &= in^* \in \mathcal{V}^* \\ outputs(a) &= out^* \in \mathcal{V}^* \\ vars(a) &= inputs(a) \cup outputs(a) \in \mathcal{V}^*\end{aligned}$$

PICWEB Illustration. the `getPictures` process invokes a key registry to retrieve the key associated to the FLICKR[®] service. This requirement step is defined as an activity a , using a *service* constant as input and a *key* variable as output.

$$\begin{aligned}a &= (a, k_a, \{service\}, \{key\}) \in \mathcal{A}, \\ service &= (service, string, false) \in \mathcal{C}, \text{ value}(service) = 'flickr' \\ key &= (key, string, false) \in \mathcal{V}\end{aligned}$$

Activity Kinds (*Kinds*). The ADORE meta-model defines nine *Kinds* of activities, classified through an inheritance graph in the object-oriented paradigm. We use closed terms (*i.e.*, terms without any free variables) to represent these concepts in first-order logic. The complete mapping between meta-model concepts and closed-terms is summarized in TAB. 4.1

Family	Concept	Term/Arity	Syntax
Business	Assign	<i>assign</i> /1	<i>assign</i> (<i>fct</i>)
—	Invoke	<i>invoke</i> /2	<i>invoke</i> (<i>srv</i> , <i>op</i>)
—	Nop	<i>nop</i> /0	<i>nop</i>
Interface	Receive	<i>receive</i> /0	<i>receive</i>
—	Reply	<i>reply</i> /0	<i>reply</i>
—	Throw	<i>throw</i> /0	<i>throw</i>
Fragment	Predecessors	\mathbb{P} /0	\mathbb{P}
—	Hook	<i>hook</i> /0	<i>hook</i>
—	Successors	\mathbb{S} /0	\mathbb{S}

Table 4.1: From activity inheritance graph to *Kind* (closed-terms) model

Concepts that do not define additional attributes are directly mapped to the associated ground term (*e.g.*, the **Nop** concept is mapped into a *nop* kind). Two concepts enhance the **Activity** meta-class with additional attributes. We represent these concepts (*i.e.*, **Assign** and **Invoke**) as closed terms, where bound variables defined inside these terms contain the additional attributes. As a consequence, the **Assign** meta-class (which holds a **function**) is mapped into a closed term of arity 1, named *assign*. We illustrate³ such a mapping in FIG. 4.6. The \mathbb{P} , \mathbb{S} and *hook* kind are defined to refer to preexisting behavior. We describe in the *fragment* paragraph how we use it to define incomplete processes which aim to be integrated into others.

³Models conforms to the ADORE meta-model are textually defined using the HUTN syntax [Rose et al., 2008].

$\text{Assign } \{\text{function: "f"}\} \rightsquigarrow \text{assign}(f)$
 $\text{Invoke } \{\text{service: "s"; operation: "o"}\} \rightsquigarrow \text{invoke}(s, o)$

Figure 4.6: “Model to Closed Term” mapping example

PICWEB Illustration. According to the requirement, the `getPictures` process retrieves the key associated to `FLICKR` in the key registry. This step is represented in ADORE using an activity $a \in \mathcal{A}$. This activity invokes the operation `getByKey` defined in the registry service.

$\text{kind}(a) = \text{invoke}(\text{registry}, \text{getByKey})$

Limitation. In ADORE, we consider activities inputs and outputs as sets of variables. According to the definition of a set, one cannot use as input (or output) the same variable twice in a given activity. For example, we consider here an *assign* activity, relying on the *add* function to perform the sum of two integers. Based on the previously defined formalism, one cannot define an activity which reify the following assignment: $x' \leftarrow \text{add}(x, x)$. At the implementation level, we define inputs (respectively outputs) as a set of ordered pairs $(\text{param}, \text{var}) \in (\text{GroundTerm} \times \mathcal{V})$, representing that the variable *var* is used to fill the parameter *param*. Assuming that the *add* function signature is $\text{add}(a, b) \mapsto a + b$, the previous example is then implemented as $x' \leftarrow \text{add}((a, x), (b, x))$. We consider these considerations as *technicals* and keep it out of the formal model to lighten the description.

4.3.3 Relations (\mathcal{R})

Relations defined in ADORE model activity scheduling. We define a binary relation⁴ as an ordered pair of activities (i.e., *left* and *right*). We denote as $\text{left} < \text{right}$ the fact that a relation exists between *left* and *right*. As ADORE defines different kinds of relations, we use a *label* (defined as a closed term) to reify this information.

$$\text{left} < \text{right} \Rightarrow \exists r = (\text{left}, \text{right}, \text{label}) \in (\mathcal{A} \times \mathcal{A} \times \text{ClosedTerm}) = \mathcal{R}$$

Path (i.e., relation transitive closure). We define a *path* between two activities a and a' as the set of related activities used to go from a to a' according to a given set of relations $R \in \mathcal{R}^*$. The function path^+ (defined as *path* extension) computes all the existing paths between a and a' .

$$\begin{aligned}
 \text{path} : \mathcal{R}^* \times \mathcal{A} \times \mathcal{A} &\rightarrow \mathcal{A}^* \\
 (R, a, a') &\mapsto a^* \equiv \begin{cases} a < a' \in R \Rightarrow a^* = \{a, a'\} \\ \wedge \exists a < \alpha \in R, \text{path}(R, \alpha, a') = A \Rightarrow a^* = \{a\} \cup A \end{cases} \\
 \text{path}^+ : \mathcal{R}^* \times \mathcal{A} \times \mathcal{A} &\rightarrow \{\mathcal{A}^*\}^* \\
 (R, a, a') &\mapsto \{A^* \mid \forall p = \text{path}(R, a, a'), p \in A^*\}
 \end{aligned}$$

To access *relation* content, we define a set of eponymous functions:

$$\begin{aligned}
 \forall \text{rel} = (l, r, \text{lab}) &\in \mathcal{R} \\
 \text{left}(\text{rel}) &= l \in \mathcal{A} \\
 \text{right}(\text{rel}) &= r \in \mathcal{A} \\
 \text{label}(\text{rel}) &= \text{lab} \in \text{ClosedTerm}
 \end{aligned}$$

In this section, we only describe the semantics of each relation *in isolation*, considering only two activities. The way relations interact to produce complex behavior (and the associated execution semantics) is described in SEC. 5.3.

⁴ADORE uses only binary relations.

Wait For ($a \prec a'$). The *waitFor* relation defines a basic wait between two activities. Let $a \prec a' \in \mathcal{R}$. The activity a' will start only after the end of the activity a .

$$a \prec a' \equiv (a, a', \text{waitFor}) \in \mathcal{R}$$

PICWEB Illustration. According to the `getPictures` requirement description, the process invokes the key registry ($a_1 \in \mathcal{A}$) and *then* invokes the FLICKR[®] service with the retrieved key ($a_2 \in \mathcal{A}$). This wait is denoted as $a_1 \prec a_2$.

Guards ($a \stackrel{c}{\prec} a'$, $a \stackrel{\neg c}{\prec} a'$). The *guard* relation allows one to define conditional branches. It enforces the *waitFor* relation to restrict the execution of its *right* activity according to the boolean response of the *left* one. We use a closed term $\text{guard}(\text{variable}, \text{value})$ to model this information. The ground term *variable* is the name of an existing boolean variable, and the *value* is bound to a boolean value. Since $a \stackrel{c}{\prec} a'$, the a' activity will start only if the end of activity a assigns the boolean *true* to the variable c .

$$\begin{aligned} a \stackrel{c}{\prec} a' &\equiv (a, a', \text{guard}(c, \text{true})) \in \mathcal{R}, c \in \text{outputs}(a) \wedge \text{type}(c) = \mathbb{B} \\ a \stackrel{\neg c}{\prec} a' &\equiv (a, a', \text{guard}(c, \text{false})) \in \mathcal{R}, c \in \text{outputs}(a) \wedge \text{type}(c) = \mathbb{B} \end{aligned}$$

Remark on the *guard* notation. To lighten the logical description used in this document, and unless it is explicitly contradicted, we use the $a \stackrel{c}{\prec} a'$ notation to refer to the existence of a guard, and the $a \stackrel{\neg c}{\prec} a'$ notation to refer to its opposite.

PICWEB Illustration. For performance reasons, one can add a cache mechanism in front of the FLICKR[®] invocation in the `getPictures` process. As a consequence, the FLICKR[®] service will be invoked (a_1) only if the *isValid* operation of the *cache* service returns *false* (stored in a boolean variable *val*) when invoked (*test*): $\text{test} \stackrel{\neg \text{val}}{\prec} a_1$.

Weak Wait ($a \ll a'$). The *weakWait* relation defines a potential disjunction between the predecessors of the *right* activity. This relation allows designers to explicitly define *alternative* paths in their process. As a consequence, these relations make sense when combined with each others (see SEC. 5.4 for more details on the execution semantic associated to the *weakWait* relations). Considering $\{a' \ll a, a'' \ll a\} \in \mathcal{R}^2$, the activity a will start after the end of activity a' *or* after the end of activity a'' . A single *weakWait* is equivalent to a *waitFor* relation.

$$\left\{ \begin{array}{l} a' \ll a \equiv (a', a, \text{weakWait}) \in \mathcal{R} \\ a'' \ll a \equiv (a'', a, \text{weakWait}) \in \mathcal{R} \end{array} \right\} \Rightarrow a \text{ waits the end of } a' \text{ or } a''$$

$$a' \ll a \in \mathcal{R}, \nexists a'' \ll a \in \mathcal{R} \Rightarrow a' \prec a$$

PICWEB Illustration. The FLICKR[®] service may take too much time to provide a response when invoked ($a_1 \in \mathcal{A}$) in the `getPictures` process. To avoid such a behavior, one can start a stopwatch in parallel ($a_2 \in \mathcal{A}$). The activity a_3 following the FLICKR[®] invocation (which reply the result) will then start after the end of a_1 (normal execution) *or* after the end of a_2 (timeout triggered). This behavior is modeled in ADORE with the two following relations: $\{a_1 \ll a_3, a_2 \ll a_3\}$

On Failure ($a \stackrel{f}{\blacktriangleleft} a'$). The *onFailure* relation allows one to catch a fault thrown by an activity during its execution. As a consequence, $a \stackrel{f}{\blacktriangleleft} a'$ means that the activity a' will start if a fault f is thrown by the activity a .

$$a \stackrel{f}{\blacktriangleleft} a' \equiv (a, a', \text{onFailure}(f)) \in \mathcal{R}$$

PICWEB Illustration. During its execution, the FLICKR[®] service can throw an “Invalid API Key” fault (abbreviated as *iAK*) if the given *key* is not a valid one (e.g., revoked, outdated) when performing the invocation (a_1). One can decide to store such an error in a log using a dedicated invocation (a_2). The following relation represents this behavior: $a_1 \stackrel{iAK}{\blacktriangleleft} a_2$.

Paths specialization. The previously defined *path* function can be specialized into two different categories of paths:

- a *control-path* only involves *waitFor*, *weakWait* and *guards* relations.
- an *exceptional-path* involves at least one *onFailure* relation.

4.3.4 Business Processes (\mathcal{P})

A business process $p \in \mathcal{P}$ is defined as a *name* (defined as a ground term), a set of variables *vars* to exchange data between service invocations, a set of activities *acts* which manipulate the variables, and a set of relations *rels* to schedule the activity set according to a partial order.

$$p = (name, vars, acts, rels) \in (Symbol \times \mathcal{V}^* \times \mathcal{A}^* \times \mathcal{R}^*) = \mathcal{P}$$

To access *process* content, we define a set of eponymous functions:

$$\begin{aligned} \forall p = (n, v^*, a^*, r^*) &\in \mathcal{P} \\ name(p) &= n \in GroundTerm \\ vars(p) &= v^* \in \mathcal{V}^* \\ acts(p) &= a^* \in \mathcal{A}^* \\ rels(p) &= r^* \in \mathcal{R}^* \end{aligned}$$

Relation to Graph Theory. Let $p \in \mathcal{P}$. The previously defined formalism is definitively inspired by graph theory. We use the following definition of *directed graphs*⁵ to draw a parallel between the ADORE meta-model and graph theory:

“A **directed graph** (or just **digraph**) D consists of a non-empty finite set $V(D)$ of elements called **vertices** and a finite set $A(D)$ of ordered pairs of distinct vertices called **arcs**. A **weighted directed (pseudo)graph** is a directed graph D along with a mapping $c : A(D) \rightarrow R$. Thus, a weighted directed (pseudo)graph is a triple $D = (V(D), A(D), c)$. If a is an element (i.e., an arc) of a weighted directed (pseudo)graph $D = (V(D), A(D), c)$, then $c(a)$ is called the **weight** or the **cost** of a .”

[Bang-Jensen and Gutin, 2000]

Considering activities as vertices ($acts(p) \equiv V(D)$) and relations as arcs ($rels(p) \equiv A(D)$), we can define a process as a directed graph. To include the different relations available in ADORE, we use the *cost* function associated to a weighted digraph, and consider the *cost* of an arc as the *label* associated to its relation.

$$c : A(D) \rightarrow R \equiv label : \mathcal{R} \rightarrow ClosedTerm$$

Orchestrations \neq Fragments. According to the previously defined inheritance graph (see FIG. 4.1), a business process can be either an orchestration or a fragment of orchestration. We define two boolean functions named *isOrchestration?* and *isFragment?* to model this information.

- *isOrchestration?* : $\mathcal{P} \rightarrow \mathbb{B}$
- *isFragment?* : $\mathcal{P} \rightarrow \mathbb{B}$

⁵In the context of ADORE, we do not differentiate *direct graphs*, *pseudographs* and *multigraph*, according to PROP. 4.9

PICWEB Illustration. We consider here the `truncate` business process, which truncates a received set of URLs according to a user-given threshold. It is composed by five activities which respectively (a_0) receives the user given parameters, (a_1) counts the cardinality of the received set, (a_2) tests if the set must be truncated, (a_3) performs the truncation and finally (a_4) replies the truncated set. These activities are using four variables to perform such a goal: url^* the set to restrict, $threshold$ the user-given value, $length$ the effective cardinality of the received set and c a boolean condition deciding if the set must be truncated or not. An anonymous constant λ_0 is used to store the 1 value, used when truncating the set. This process involves *waitFor* and *guards* relations to model the expected behavior. A complete description of this process is depicted in FIG. 4.7, expressed using the ADORE formal model.

$$\begin{aligned}
url^* &= (url, string, true) \in \mathcal{V} \\
threshold &= (threshold, integer, false) \in \mathcal{V} \\
length &= (length, integer, false) \in \mathcal{V} \\
c &= (c, boolean, false) \in \mathcal{V} \\
\lambda_0 &= (\lambda_0, integer, false) \in \mathcal{C}, value(\lambda_0) = '1' \\
V &= \{url^*, threshold, length, c, \lambda_0\} \in \mathcal{V}^* \\
a_0 &= (a_0, receive, \emptyset, \{url^*, threshold\}) \in \mathcal{A} \\
a_1 &= (a_1, assign(count), \{url^*\}, \{length\}) \in \mathcal{A} \\
a_2 &= (a_2, assign(isGreaterThan), \{length, threshold\}, \{c\}) \in \mathcal{A} \\
a_3 &= (a_3, assign(subset), \{url^*, \lambda_0\}, \{url^*\}) \in \mathcal{A} \\
a_4 &= (a_4, reply, \{url^*\}, \emptyset) \in \mathcal{A} \\
A &= \{a_0, a_1, a_2, a_3, a_4\} \in \mathcal{A}^* \\
R &= \{a_0 \prec a_1, a_1 \prec a_2, a_2 \stackrel{c}{\prec} a_3, a_2 \stackrel{\neg c}{\prec} a_4, a_3 \prec a_4\} \in \mathcal{R}^* \\
truncate &= (truncate, V, A, R) \in \mathcal{P}
\end{aligned}$$

Figure 4.7: `truncate` business process modeled using ADORE formalism

Fragments (denoted as \mathcal{F}) and orchestrations (denoted as \mathcal{O}) represent a partition⁶ of \mathcal{P} . These two sets can be defined in intention:

$$\begin{aligned}
\mathcal{O} &\equiv \{p \in \mathcal{P} \mid isOrchestration?(p)\} \\
\mathcal{F} &\equiv \{p \in \mathcal{P} \mid isFragment?(p)\} \\
\mathcal{P} &= \mathcal{O} \cup \mathcal{F}, \quad \mathcal{O} \cap \mathcal{F} = \emptyset
\end{aligned}$$

On the one hand, an orchestration defines the behavior of an *operation*, exposed in a *service*. This additional information is modeled as a closed term *orch* of arity 2, available through a dedicated function (*extras*).

$$\begin{aligned}
extras : \mathcal{O} &\rightarrow ClosedTerm \\
o &\mapsto orch(service, operation)
\end{aligned}$$

On the other hand, fragments represent incomplete behavior (which aims to be integrated into other processes) and consequently defines *ghosts* variables (handled in the fragment but concretely defined in the targeted business process). These variables are reified through fragment *hook* activity, which makes the link between the fragment and the targeted process. We define a dedicated function *ghost* : $\mathcal{F} \rightarrow \mathcal{V}^*$ to identify such variables. To reify activities existing in the targeted process, we use the \mathbb{P} (predecessors), \mathbb{S} (successors) and *hook* activity kind. We

⁶A partition is defined as the following: “Given a set S , a partition is a pairwise disjoint family $P = \{A_i\}$ of non-empty subsets of S whose union is S .” [Rosen, 2000]

also provide a set of eponymous functions to easily access to these artifacts in a given process (according to PROP. 4.13 —fragment coherence, a fragment can only define one *hook* —respectively \mathbb{P} and \mathbb{S} — activity).

$$\begin{aligned}
\mathbb{P} : \mathcal{F} &\rightarrow \mathcal{A} \\
f &\mapsto a \in \text{acts}(f), \text{kind}(a) = \mathbb{P} \\
\text{hook} : \mathcal{F} &\rightarrow \mathcal{A} \\
f &\mapsto a \in \text{acts}(f), \text{kind}(a) = \text{hook} \\
\mathbb{S} : \mathcal{F} &\rightarrow \mathcal{A} \\
f &\mapsto a \in \text{acts}(f), \text{kind}(a) = \mathbb{S} \\
\text{ghosts} : \mathcal{F} &\rightarrow \mathcal{V}^* \\
f &\mapsto \text{vars}(\text{hook}(f))
\end{aligned}$$

PICWEB Illustration. As explained above, *truncate* and *getPictures* business processes are modeled as orchestrations. Additional extensions such as *cache* or *timeout* mechanisms are defined as fragments.

$$\begin{aligned}
\text{truncate} &\in \mathcal{O}, \text{extras}(\text{truncate}) = \text{orch}(\text{helpers}, \text{truncate}) \\
\text{getPictures} &\in \mathcal{O}, \text{extras}(\text{getPictures}) = \text{orch}(\text{picweb}, \text{getPictures}) \\
\text{cache} &\in \mathcal{F}, \text{timeout} \in \mathcal{F}
\end{aligned}$$

Path syntactic shortcut. For a given process $p \in \mathcal{P}$, we refer to the previously defined *path* function according to the $\text{rels}(p)$ set of relations, and activities contained in the same process. Given a and a' two activities contained in $\text{acts}(p)$, we denote as $a \rightarrow a'$ the existence of a path between a and a' . To lighten the expression of properties on the ADORE metamodel, we also consider that $a \rightarrow a'$ can be considered as $a = a'$.

$$a \rightarrow a' \equiv \exists p \in \mathcal{P}, a \in \text{acts}(p), a' \in \text{acts}(p), \text{path}(\text{rels}(p), a, a') \neq \emptyset \vee a = a'$$

Following the same convention, we denote as \rightarrow a “control-path” and \rightarrow an “exceptional path”.

Entry & Exit points. According to their kinds and their positions in the partial order induced by the relations, activities can be defined as entry or exit points.

$$\begin{aligned}
\text{entry} : \mathcal{P} &\rightarrow \mathcal{A} \\
p &\mapsto a \in \text{acts}(p), \text{kind}(a) \in \{\text{receive}, \mathbb{P}\} \\
&\quad \wedge \nexists a' \in \text{acts}(p), a' \rightarrow a \\
\text{exit} : \mathcal{P} &\rightarrow \mathcal{A}^* \\
p &\mapsto \{a \mid a \in \text{acts}(p), \text{kind}(a) \in \{\text{reply}, \text{throw}, \mathbb{S}\}\} \\
&\quad \wedge \nexists a' \in \text{acts}(p), a \rightarrow a'\}
\end{aligned}$$

4.3.5 Universe (\mathcal{U})

An universe $u \in \mathcal{U}$ is defined as a set of business processes $\text{procs} \in \mathcal{P}^*$ (simple renaming).

$$u = \text{procs} \in \mathcal{P}^* = \mathcal{U}$$

PICWEB Illustration. According to the previous section, the universe associated to PICWEB is composed by four processes.

$$\text{picweb} = \{\text{truncate}, \text{getPictures}, \text{cache}, \text{timeout}\}$$

To access *universe* content, we define a *procs* function, and several transitive relations to reach entities defined in the contained processes.

$$\begin{aligned}
\forall u = p^* &\in \mathcal{U} \\
procs(u) &= p^* \in \mathcal{P}^* \\
orchs(u) &= \{p \mid p \in procs(u), isOrchestration?(p)\} \in \mathcal{O}^* \\
frags(u) &= \{p \mid p \in procs(u), isFragment?(p)\} \in \mathcal{F}^* \\
vars(u) &= \bigcup_{p \in p^*} vars(p) \in \mathcal{V}^* \\
acts(u) &= \bigcup_{p \in p^*} acts(p) \in \mathcal{A}^* \\
rels(u) &= \bigcup_{p \in p^*} rels(p) \in \mathcal{R}^*
\end{aligned}$$

4.3.6 Iteration Policies (\mathcal{I}) & Blocks

An iteration policy $i \in \mathcal{I}$ is defined as a data-set variable set^* , a scalar variable $scalar$, a *kind* and a set of involved activities $acts$. The semantics associated to iteration policies can be informally described as follows: activities contained in the activity set will be executed for each $scalar \in set^*$, according to its *kind*. The complete semantics associated to iteration policies and their *kinds* is described in SEC. 5.5. Even if *policies* are not contained by a process as they represent meta-data associated to ADORE entities, we consider as part of the *process* concept a function $pols : \mathcal{P} \rightarrow \mathcal{I}^*$ which returns all the policies involved in a given process.

$$i = (set^*, scalar, kind, acts) \in (\mathcal{V} \times \mathcal{V} \times \{serial, parallel\} \times \mathcal{A}^*) = \mathcal{I}$$

To access *iteration policies* content, we define a set of eponymous functions:

$$\begin{aligned}
\forall i = (s, v, k, a^*) &\in \mathcal{I} \\
set(i) &= s \in \mathcal{V}, isSet?(s) \\
scalar(i) &= v \in \mathcal{V}, \neg isSet?(v) \\
kind(i) &= k \in \{serial, parallel\} \\
acts(i) &= a^* \in \mathcal{A}^*
\end{aligned}$$

Blocks. Iteration policies rely on a *loop* semantic⁷. As a consequence, the set of activities involved in an iteration policy must respect additional constraints to be semantically valid. We define the concept of *block* of activities to reify such a restriction. A block is basically defined as a set of activities, equipped with functions defined to handle it.

We define the “*boundary*” of a block b through its *interface*, that is, activities involved in the block and related to others which are not part of the block. The reciprocal function (*internal*) gives access to the internal activities of the block, that is, activities which do not communicate with others defined outside the block. We require that a block of activities involved in an iteration policy must not contain any “*interface*” activity (*i.e.*, *receive*, *reply* & *throw*). Based on this assumption, we ensure that a block is always preceded (repectively succeeded) by at least one activity. The predecessors (*preds*) of a given block b (respectively the successors *succs*) are defined as the activities which immediately precede the first activities of b (respectively the last ones). A block is considered as a *singleton* when it only contains a single activity.

$$\begin{aligned}
\text{Let } p \in \mathcal{P} &\quad a^* \subset acts(p) \\
firsts : \mathcal{A}^* &\rightarrow \mathcal{A}^* \\
a^* &\mapsto \{a \mid a \in a^*, \exists a' < a \in rels(p) \wedge a' \notin a^*\} \\
lasts : \mathcal{A}^* &\rightarrow \mathcal{A}^* \\
a^* &\mapsto \{a \mid a \in a^*, \exists a < a' \in rels(p) \wedge a' \notin a^*\}
\end{aligned}$$

⁷ as explained in SEC. 5.5

$$\begin{aligned}
\text{interface} : \mathcal{A}^* &\rightarrow \mathcal{A}^* \\
a^* &\mapsto \text{firsts}(a^*) \cup \text{lasts}(a^*) \\
\text{internal} : \mathcal{A}^* &\rightarrow \mathcal{A}^* \\
a^* &\mapsto \{a \mid \exists f \in \text{firsts}(a^*), a \in a^*, f \rightarrow a \wedge a \notin \text{lasts}(a^*)\} \\
\text{preds} : \mathcal{A}^* &\rightarrow \mathcal{A}^* \\
a^* &\mapsto \{a' \mid \exists a \in a^*, \exists a' < a \in \text{rels}(p) \wedge a' \notin a^*\} \\
\text{nexts} : \mathcal{A}^* &\rightarrow \mathcal{A}^* \\
a^* &\mapsto \{a' \mid \exists a \in a^*, \exists a < a' \in \text{rels}(p) \wedge a' \notin a^*\} \\
\text{singleton?} : \mathcal{A}^* &\rightarrow \mathbb{B} \\
a^* &\mapsto |a^*| = 1, \text{firsts}(a^*) = \text{lasts}(a^*) = \text{interface}(a^*) = \text{internal}(a^*)
\end{aligned}$$

The activities involved in an iteration must define a *well-formed* block of activities, that is, a block of activities which respects PROP. 4.21.

Limitations. Regarding the current ADORE meta-model, it is not possible to use several iteration policies on the same activity. Such an extension is one of the perspectives of this work. ADORE also does not support policies defined inside others.

4.4 Properties

We present in this section a set of *properties* defined over the previously defined formal concepts. Without further explanations, an ADORE model (*i.e.*, a model conforms to the ADORE meta-model) assumes the respect these properties.

Objectives. These properties reify constraints expressed over the meta-model and consequently restrict the ADORE expressiveness into entities compatible with the composition mechanisms described in PART III. At the implementation level, the composition engine throws an exception when it identifies a constraint violation.

4.4.1 Variables

Property 4.1 (variable uniqueness). *Let $v \in \mathcal{V}$ a variable. There is no other variables which use the same name. This property allow the assimilation of a variable and its name while performing a composition, and avoid inadvertent capture [Kohlbecker et al., 1986].*

$$\forall v \in \mathcal{V}, \forall v' \in \mathcal{V}, \text{name}(v) = \text{name}(v') \Rightarrow v = v'$$

Property 4.2 (constant immutability). *Let $c \in \mathcal{C}$ a constant. There is no activity $a \in \mathcal{A}$ which uses c as an output.*

$$\forall c \in \mathcal{C}, \nexists a \in \mathcal{A}, c \in \text{outputs}(a)$$

4.4.2 Activities

Property 4.3 (activity uniqueness). *Let $a \in \mathcal{A}$ an activity. There is no other activity which use the same name. This property allow the assimilation of an activity and its name while performing a composition.*

$$\forall a \in \mathcal{A}, \nexists a' \in \mathcal{A}, a \neq a' \wedge \text{name}(a) = \text{name}(a')$$

Property 4.4 (Restriction on orchestration activities). *Let $a \in \mathcal{A}$ an activity. Receive activities can only use output variables (as value are received from the outside and then assigned to local variables) and reply activities can only use input variables (as the content of these variables*

will be responded to the outside). A throw activity only accepts one input variable of type string which contains the fault name.

$$\begin{aligned} \forall a \in \mathcal{A}, \text{kind}(a) = \text{receive} &\Rightarrow \text{inputs}(a) = \emptyset \\ \forall a \in \mathcal{A}, \text{kind}(a) = \text{reply} &\Rightarrow \text{outputs}(a) = \emptyset \\ \forall a \in \mathcal{A}, \text{kind}(a) = \text{throw} \\ &\Rightarrow \text{outputs}(a) = \emptyset \wedge \text{inputs}(a) = \{v\}, \text{type}(v) = \text{string} \end{aligned}$$

Property 4.5 (Restriction on fragment activities). Let $u \in \mathcal{U}$, and $f \in \text{frags}(u)$ a fragment. As predecessors and successors potentially represent multiple activities dealing with multiple variables, they cannot hold any variable from the fragment point of view.

$$\forall a \in \mathcal{A}, \text{kind}(a) \in \{\mathbb{P}, \mathbb{S}\} \Rightarrow \text{vars}(a) = \emptyset$$

4.4.3 Relations

Property 4.6 (relation containment). A relation contained in a (unique) process $p \in \mathcal{P}$ involves activities from the same process.

$$\forall a < a' \in \mathcal{R}, \exists! p \in \mathcal{P}, a < a' \in \text{rels}(p) \Rightarrow a \in \text{acts}(p) \wedge a' \in \text{acts}(p)$$

Property 4.7 (irreflexive relations). Relations in ADORE reify waits between activities. As it does not make any sense for an activity to wait for its own end, all relations defined in ADORE are defined as irreflexive.

$$\forall a \in \mathcal{A}, a < a \notin \mathcal{R}$$

Property 4.8 (asymmetric relations). According to the same reasons as the irreflexive property, direct cycle such as $\{a < a', a' < a\}$ cannot be defined in ADORE.

$$\forall (a, a') \in \mathcal{A}^2, a < a' \in \mathcal{R} \Rightarrow a' < a \notin \mathcal{R}$$

Property 4.9 (acyclic relations). This property is a generalization of the previous one. It ensures that there is no existing cycle in the relation set. As a consequence, a process $p \in \mathcal{P}$ is assimilated as a directed acyclic graph (DAG).

$$\forall (a, a') \in \mathcal{A}^2, a < a' \in \mathcal{R} \Rightarrow a' \rightarrow a \notin \mathcal{R}$$

Property 4.10 (guard vivacity). To ensure that guards relations are always semantically valid, no other activity may change the value assigned in the boolean condition while a guard can be crossed.

$$\forall (a, a') \in \mathcal{A}^2, \forall \alpha \in \mathcal{A}, \exists a' \prec a' \in \mathcal{R} \wedge c \in \text{outputs}(\alpha) \Rightarrow \alpha \rightarrow a$$

4.4.4 Business Process

Property 4.11 (single entry-point). Let $p \in \mathcal{P}$. There is only one unique entry point activity $a \in \text{acts}(p)$ defined in p .

$$\forall p \in \mathcal{P}, \exists! a \in \text{acts}(p), a = \text{entry}(p)$$

Property 4.12 (“entry-to-exit” path existence). Let $p \in \mathcal{P}$. All exit points of p are connected to its entry point through a path.

$$\forall p \in \mathcal{P}, \forall a \in \text{acts}(p), a = \text{exit}(p) \Rightarrow \text{entry}(p) \rightarrow a$$

Property 4.13 (fragment structure). Let $f \in \mathcal{F}$ a fragment. This fragment can only define one predecessor activity (\mathbb{P}), one hook activity and one successors activity (\mathbb{S}).

$$\begin{aligned} \forall f \in \mathcal{F}, \exists! a \in \text{acts}(f), a = \mathbb{P}(f) \\ \forall f \in \mathcal{F}, \exists! a \in \text{acts}(f), a = \text{hook}(f) \\ \forall f \in \mathcal{F}, \exists! a \in \text{acts}(f), a = \mathbb{S}(f) \end{aligned}$$

Property 4.14 (fragment coherence). Let $f \in \mathcal{F}$ a fragment. This fragment defines at least one path which connects predecessors, hook and successors.

$$\forall f \in \mathcal{F}, \mathbb{P}(f) \rightarrow \text{hook}(f) \wedge \text{hook}(f) \rightarrow \mathbb{S}(f)$$

Property 4.15 (process isolation). Let $p \in \mathcal{P}$ a process. Activities and variables contained in p are locally scoped and as a consequence cannot be shared with other processes.

$$\begin{aligned} \forall p \in \mathcal{P}, \forall v \in \text{vars}(p) &\Rightarrow \nexists p' \in \mathcal{P}, p' \neq p \wedge v \in \text{vars}(p') \\ \forall p \in \mathcal{P}, \forall a \in \text{acts}(p) &\Rightarrow \nexists p' \in \mathcal{P}, p' \neq p \wedge a \in \text{acts}(p') \end{aligned}$$

Property 4.16 (activity path completeness). All activities defined in a process $p \in \mathcal{P}$ must be connected to the entry point and to at least one exit point.

$$\forall p \in \mathcal{P}, \forall a \in \text{acts}(p), a \neq \text{entry}(p), a \notin \text{exit}(p) \Rightarrow \exists e \in \text{exit}(p), \text{first}(p) \rightarrow a \rightarrow e$$

Property 4.17 (activity position consistency). An activity defined as a receive or a \mathbb{P} cannot be preceded by another activity (i.e., it must be an entry point). Respectively, an activity defined as a reply, a throw or a \mathbb{S} cannot be followed by another activity (i.e., it must be an exit point).

$$\begin{aligned} \forall p \in \mathcal{P}, \forall a \in \text{acts}(p), \text{kind}(a) \in \{\text{receive}, \mathbb{P}\} &\Rightarrow a = \text{entry}(p) \\ \text{kind}(a) \in \{\text{reply}, \text{throw}, \mathbb{S}\} &\Rightarrow a \in \text{exit}(p) \end{aligned}$$

4.4.5 Iteration Policies & Block

Property 4.18 (dataset isolation). Let $i \in \mathcal{I}$ an iteration policy. The associated data set $\text{set}(i)$ cannot be modified directly by an activity involved in the iteration.

$$\forall i \in \mathcal{I}, \forall a \in \text{acts}(i), \text{set}(i) \notin \text{outputs}(a)$$

Property 4.19 (no cross-boundary fault handling). Let $i \in \mathcal{I}$ an iteration policy. As the activities involved in i will be executed multiple times, we require that no *onFailure* relation can cross an iteration policy boundary.

$$\forall i \in \mathcal{I}, \forall a \in \text{acts}(i) \subset \text{acts}(p), \nexists a \xrightarrow{f} a' \in \text{rels}(p) \wedge a' \notin \text{acts}(i)$$

Property 4.20 (block consistency). We consider a block of activities $b \in \mathcal{A}^*$ as consistent iff all activities defined inside are contained by the same process $p \in \mathcal{P}$.

$$\forall b \in \mathcal{A}^*, \forall a \in b, \exists p \in \mathcal{P}, a \in \text{acts}(p) \Rightarrow \nexists a' \in b, a' \notin \text{acts}(p)$$

Property 4.21 (well-formed activity block). Let $b \in \mathcal{A}^*$ a consistent block of activities. We consider a block as well-formed iff (i) it is defined as a singleton or (ii) its interface activities do not overlap its internal content (that is, $\text{internal}(b) \cap \text{external}(b) = \emptyset$). In other words, there is no activity defined in b which is related to activities defined inside and outside b (excepting for the singleton block).

Equipping ADORE: Syntax & Semantics

«He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast.»

Leonardo da Vinci

Contents

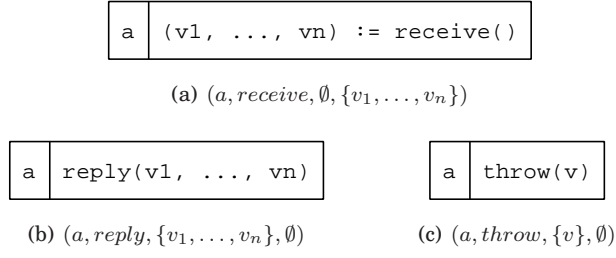
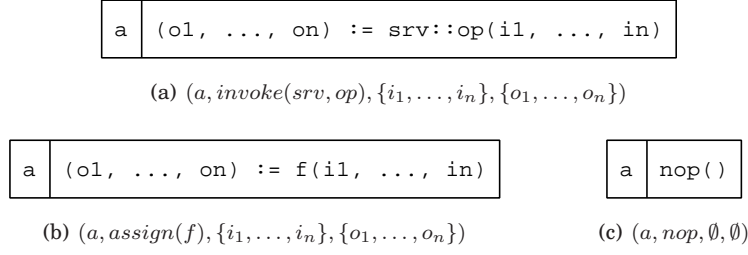
5.1 Introduction	51
5.2 Graphical Syntax	51
5.2.1 Representation of Activities	52
5.2.2 Representation of Relations	53
5.2.3 Iteration Policy	53
5.2.4 Process Representation	53
5.3 Execution Semantics	54
5.3.1 Activity Lifecycle Automaton	56
5.3.2 Process Execution & Activity Triggering	57
5.3.3 Iteration Policies Handling	57
5.4 From Relations to Trigger Formula (φ)	58
5.4.1 Process Entry Point & Single Predecessors	58
5.4.2 Composition of Relations	58
5.5 Iteration Policy Handling	65

5.1 Introduction

In CHAP. 4, we defined a formal model to reify ADORE foundations. The goals of this chapter are multiples. We start by defining a graphical syntax associated to the formal model in SEC. 5.2, to lighten the expression of ADORE models. Then, we define an execution semantic for ADORE artifacts in SEC. 5.3. Finally, we expose how multiple relations can be composed to express complex behavior (SEC. 5.4). SEC. 5.5 is dedicated to iteration policy semantic description.

5.2 Graphical Syntax

To make ADORE artifacts more readable, we define a graphical syntax associated to the formal model. The ultimate goal of this syntax is to allow one to easily understand the behavior of a given process. As ADORE is inspired by graph-theory, we use a graph-based representation. We basically represent activities as boxes, and relations between activities as arrows. We voluntarily choose to not represent variable declaration in the graphical syntax, according to our goal (understand the behavior).

Figure 5.1: Graphical notation associated to *interface* activitiesFigure 5.2: Graphical notation associated to *business* activities

5.2.1 Representation of Activities

We represent an ADORE activity as a box. The left part of the box contains the *name* of the activity. The right part of the box is dedicated to activity *content*, according to its kind and variable usage:

- Variables are represented by their *name*,
 - Data-sets ($v \in \mathcal{V}, isSet?(v)$) are postfixed with a $*$,
 - Constants ($c \in \mathcal{C}$) are valued and surrounded by quotes.
- Sets of variables are represented using a parenthesized form (v_1, \dots, v_n) ,
- The $:=$ symbol denotes assignment,
- The $srv:op$ notation denotes an invocation of the operation op exposed by the service srv ,
- Special keywords are dedicated to their associated kind: `receive`, `reply`, `nop` & `throw`,
- Assignments are represented using their *function* as keyword.

The notation used for *interface* activities is depicted in FIG. 5.1, and the one associated to *business* activities is depicted in FIG. 5.2.

Fragment-Specific Activities. Fragments of processes define three special activities: (i) a *hook*, (ii) its *predecessors* and (iii) its *successors*. These activities refer to the behavior of the process where the fragment will be integrated to. We decide to represent these activities using a *dashed* notation instead of a plain one, to make the difference of their intrinsic nature explicit.

Short Notation. When the *contents* of an activity is not important for a given context, we allow the usage of simple *boxes* to represent an activity, without the *contents* right part. We use this syntactic shortcut to illustrate ADORE capabilities when the activity kind is not relevant (e.g. activity scheduling). An example is represented in FIG. 5.4.

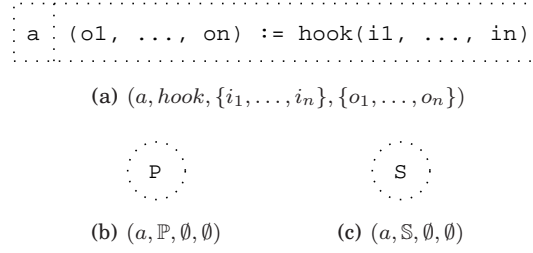
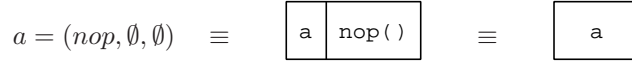
Figure 5.3: Graphical notation associated to *fragment* activities

Figure 5.4: Syntactic shortcut used to lighten graphical representation

5.2.2 Representation of Relations

We represent $a < a' \in \mathcal{R}$ a relation between two activities as an arrow starting from a and ending in a' . The ADORE meta-model defines four types of relations between activities. The *label* associated to a relation is represented by the graphic style applied to the arrow:

- a *plain* arrow represents *waitFor* relations,
- a *plain* arrow with an associated label represents a *guard*,
- a *dashed* arrow with a *round* tail represents a *weakWait*,
- a *red* arrow with a *diamond* tail represents a *onFailure* relation.

Example of these graphical notations can be found in FIG. 5.5

5.2.3 Iteration Policy

We illustrate in FIG. 5.6 the syntax associated to the iteration policy concept. We use a nested *sub-graph* notation, where activities represented inside the sub-graph are part of the activities involved in the iteration policy. The *kind* associated to the policy is represented enclosed by $\langle \rangle$. We use a “ x in x^* ” label defined inside the subgraph to represent an iteration policy I_p where $scalar(I_p) = x$ and $set(I_p) = x^*$.

5.2.4 Process Representation

We represent a *process* as a directed acyclic graph. Orchestrations and fragments differ according to their bottom label: an *orchestration* label is composed by the *extras* informations han-

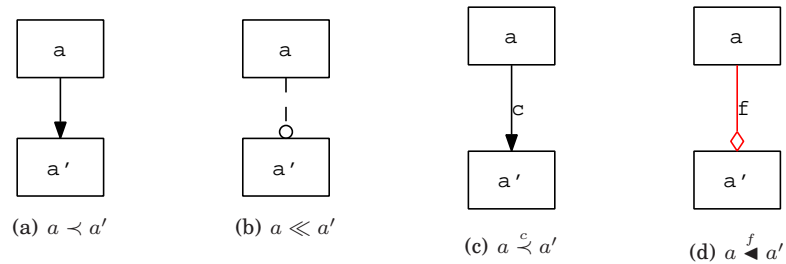


Figure 5.5: Graphical representation associated to ADORE relations

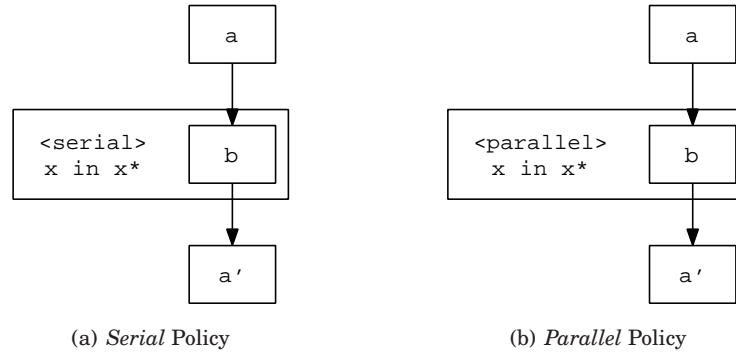


Figure 5.6: Graphical notation associated to Iteration Policies

dled by the process, and a *fragment* label contains only the name of the fragment. To ease the identification of activities provenance in composed processes, we enhance fragment with *colors*. As fragment-specific activities refer to an exterior behavior, they are not colored to emphasize the semantic difference. We use the same assumption to represent relation entering or leaving *successors* and *predecessors*, represented as *dashed* lines (as they refer to relations defined in the *target* process).

PICWEB Illustration. We depict in FIG. 5.7 artifacts extracted from the PICWEB case study. The `getPictures` process, as defined in the requirement, is represented in FIG. 5.7(a)^a. The cache fragment, used to model a cache policy, is represented in FIG. 5.7(b).

^aThis figure corresponds to the formal model represented in FIG. 4.7.

5.3 Execution Semantics

In this section, we focus on the definition of a semantics associated to the execution of modelled activities. We use this semantics in the next chapters to illustrate how the different composition algorithms interact with the original behavior during the composition. The goal of this section is (i) to describe the semantics associated to activity execution, and (ii) to illustrate how it can be instantiated for a given model.

The software architecture community defines several formal methods associated to the definition of architecture and their behavior. In the large family of Formal Architecture languages, one can use π -calculus [Sangiorgi and Walker, 2001] to model the architecture of software systems. The π -Diapason [Pourraz, 2007] approach is defined as a domain-specific π -calculus implementation, dedicated to orchestration of web services. Like the previous formalism, its goal is to check orchestration correctness by using a virtual machine in order to check its behavior.

Specification languages such as ALLOY [Jackson, 2006] can be used to express complex behavior in a software system. After the definition of **Signatures** and **Facts**, one can express **Predicates**, **Functions** and **Assertions** used to perform model-checking on the defined model. The goal of ALLOY is to check model correctness, that is, the fact that the modeled behavior respects a given program specification. ALLOY can then exhibit a counter-example if such a behavior exists in the finite scope domain used to perform the check.

Petri-Nets can be applied to model workflows [van der Aalst, 1998]. This formalism gives to workflow (i) a formal semantic, (ii) a graphical nature and (iii) a large expressiveness. Existing work on Petri-nets investigate *properties* associated to Petri-nets, as well as analysis techniques. However, according to Wil van der Aalst, “*Petri nets describing real processes tend to be complex and extremely large. Moreover, the classical Petri net does not allow for the modeling of data and time*”. As a consequence, the simple initial formalism must be enhanced into a more complex one in order to take care of these concerns.

Temporal Languages such as CCSL [Mallet et al., 2010] allow one to express **Clocks**, and a

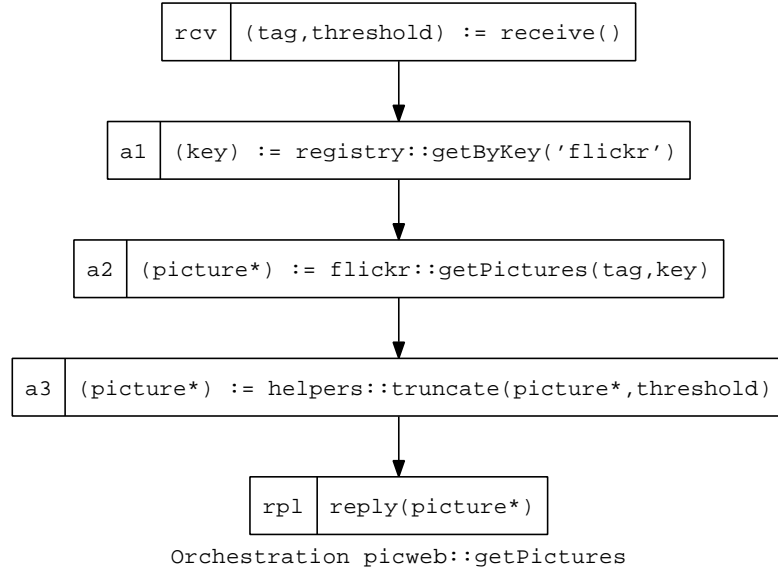
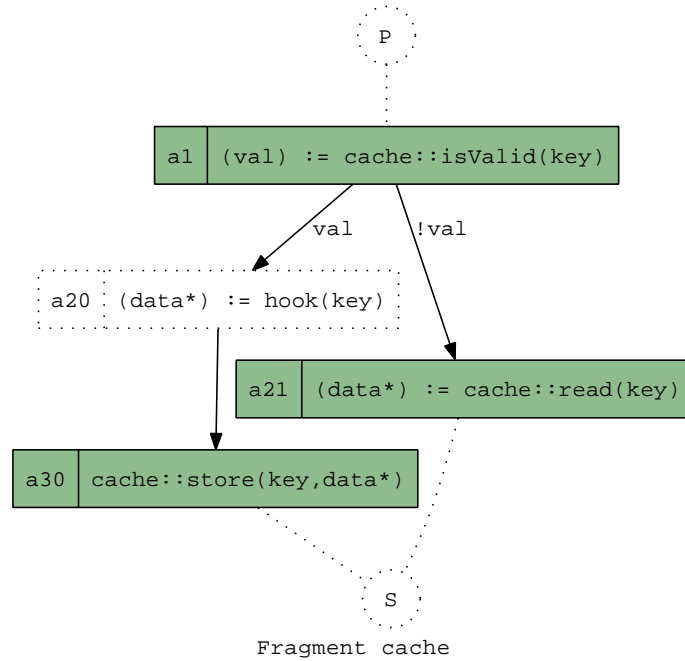
(a) `getPictures` process, modeled as an orchestration(b) `cache` concern, modeled as a fragment of process

Figure 5.7: Graphical representation illustrated on PICWEB artifacts

quasi-order between clock events through the definition of **precedence** rules and **exclusion** between pairs of activities. Based on such a temporal description of a system, it is possible to simulate the execution and to analyze traces to check the correctness of a system against its specification. From a simplified point of view, CCSL can be seen as a domain-specific model-checker for time-driven models.

Finally, process algebra and the associated temporal logic implementation (such as the automata-driven approach LTSA [Magee and Krammer, 2006]) can be used to express the behavior of a system, as an automata. Temporal properties can be checked on the automata, such as deadlocks or famine. These languages focus on concurrency description, and provide a formal foundation (based on automata) for this point of view.

Synthesis. The previously described methods focus on model-checking and correctness validation. They reason on a system *in-the-large*, and do not focus on activities *in isolation*. However, our goal is to express the execution semantics associated to *each* activity of a process, as a basis to express and demonstrate properties kept during the composition of processes. As a consequence¹, we choose a simple formalism (that is, finite automaton and boolean logic) and illustrate the semantics using such a model.

Focus on executability. The KERMETA meta-language [Fleurey, 2006] is defined to “*breathe life into meta-models*”. Using an aspect-oriented approach, the language supports the enhancement of legacy meta-models. With this approach, one can implement new operations in a meta-model, and consequently add executability concerns in a structural meta-model. The KERMETA language is used as underlying support in the CCSL tool. Using the same approach, it is possible to use KERMETA to make ADORE models executable. However, this is not our goal here. We do not want to make ADORE models executable, but instead predict their behavior in a static way. The main idea is to describe a-priori the semantic of an ADORE model based on its description, where KERMETA will support runtime reasoning, on the fly. We made the choice to support the executability of ADORE models through dedicated transformations. The first one targets the industrial BPEL standard, according to the property “One cannot ignore reality” identified in the state-of-the-art. The second (still an ongoing work) targets CCSL, for simulation and trace analysis purpose.

5.3.1 Activity Lifecycle Automaton

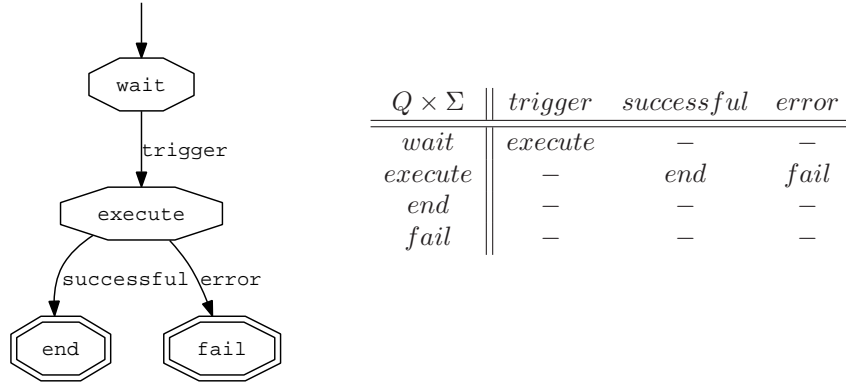
The life-cycle of an activity includes several states. Informally, an activity $a \in \mathcal{A}$ waits for its execution to begin, then executes its internal logic before ending its execution in normal or erroneous state. We use a deterministic automata to model this life-cycle.

“A deterministic finite automata consists of: (i) a finite set of states (often denoted Q), (ii) a finite set of input symbol (often denoted Σ), (iii) a transition function (often denoted δ), (iv) a start state ($q_0 \in Q$) and finally (v) a set of final states ($F \subset Q$).” [Hopcroft et al., 2001]

According to this definition, we model activity life-cycle as an automaton α which defines four states: an activity *waits* before being executed (initial state), *executes* its internal behavior, *ends* its execution (i.e., successful execution, final state) or *fails* (i.e., erroneous execution, final state). The alphabet of the automaton is as reduced as possible (*trigger*, *successful* and *error*). The transition function δ depicted in FIG. 5.8 is deterministic. As a consequence, the life-cycle automata only accepts a language $\Lambda(\alpha)$ containing two words (i.e., successful or erroneous execution).

$$\begin{aligned}
 Q &= \{wait, execute, end, fail\} \\
 \Sigma &= \{trigger, successful, error\} \\
 q_0 &= wait \\
 F &= \{end, fail\} \\
 \alpha &= (Q, \Sigma, q_0, F) \\
 \Lambda(\alpha) &= \{“trigger, successful”, “trigger, error”\}
 \end{aligned}$$

¹A bridge between ADORE and CCSL is exposed as one of the perspectives of this thesis

Figure 5.8: Activity life-cycle transition function $\delta : Q \times \Sigma \rightarrow Q$

Generalization for “Multiple Errors” Handling. According to the *service-oriented* paradigm, an operation may throw several kinds of faults. To support this feature, we extend the previously defined automata by modeling the *fail* state and the *error* symbol as closed terms of arity 1 (the fault name f). As a consequence, we can *virtually* represent an infinite number of errors, with a finite number of states. This is still valid in ADORE semantics since the important information is that the activity ends in an error state.

$$\begin{aligned}
 Q &= \{wait, execute, end, fail(f)\} \\
 \Sigma &= \{trigger, successful, error(f)\} \\
 \Lambda &= \{"trigger, successful", "trigger, error(f)"\}
 \end{aligned}$$

5.3.2 Process Execution & Activity Triggering

Each invocation of a process $p \in \mathcal{P}$ is executed as a different instance. When the process is invoked, we attach an automata to each activity defined in $acts(p)$. Each automata starts in its initial state, and waits for a *trigger* event to appear. Such a trigger is specific for each activity a , as it depends on the partial order defined by the relation set $rels(p)$. We model it as the satisfiability of a logical formula $\varphi(a)$. This formula composes the final states of a ' predecessors, according to $rels(p)$. As soon as the system can satisfy $\varphi(a)$, the *trigger* symbol is sent to the automaton associated to a . The way $\varphi(a)$ is computed according to $rels(p)$ is described in SEC. 5.4.

Process End. A process instance is considered to be *ended* since one of its *exit* point reaches a final state. The final state of the process is then defined as the final state of the reached exit point. When a process instance is *ended*, it is automatically destroyed.

5.3.3 Iteration Policies Handling

When an activity $a \in \mathcal{A}$ is involved in an iteration policy defined on a finite data-set $d^* \equiv \{d_1, \dots, d_n\}$, a must be executed for each d_i . As a consequence, we associate for each $d_i \in d^*$ an automaton α_i to a . We also enrich the *trigger* symbol of the automaton alphabet, now defined as a closed term of arity 1: $trigger(i)$. Each automaton α_i uses the $trigger(i)$ symbol in its own alphabet Σ_i . According to the same idea, the satisfaction of the formula $\varphi_i(a')$ will push the $trigger(i)$ symbol in the automaton.

$$\begin{aligned}
 \Sigma_i &= \{trigger(i), successful, error(f)\} \\
 \alpha_i &= (Q, \Sigma_i, q_o, F)
 \end{aligned}$$

5.4 From Relations to Trigger Formula (φ)

Notation. We denote as $\varphi(a)$ the *trigger formula* associated to a . This formula is defined as a boolean composition of the final states of a predecessors. We define two syntactic shortcuts $end(a)$ ² and $fail(a, f)$ ³ to reify these final states in φ . According to *guards* relations, boolean variable values may also be used. We denote as v (respectively $\neg v$) the fact that the boolean variable $v \in \mathcal{V}$ is valued with *true* (respectively *false*).

5.4.1 Process Entry Point & Single Predecessors

Entry Point. Let $p \in \mathcal{P}$ a business process. The entry-point of p starts automatically its execution, and does not need to wait for anything else. As a consequence, its trigger formula is defined as *true*, and immediately satisfied.

$$\forall a \in acts(p), a = entry(p) \Rightarrow \varphi(a) = true$$

Single Predecessors. Let $p \in \mathcal{P}$ a business process. When an activity a is preceded by a *unique* activity a' according to $rels(p)$, the associated formula only depends on the type of relations between a and a' .

- A *waitFor* implies to wait for the end of a' ,
- A *guard* implies to wait for the end of a' and check the condition value,
- An *onFailure* implies to wait for a given fault to be thrown.

$$\forall a \in acts(p), \exists! a' < a \in rels(p), \begin{cases} a' \prec a \Rightarrow \varphi(a) = end(a') \\ a' \prec^c a \Rightarrow \varphi(a) = end(a') \wedge c \\ a' \prec^e a \Rightarrow \varphi(a) = end(a') \wedge \neg c \\ a' \overset{f}{\blacktriangleleft} a \Rightarrow \varphi(a) = fail(a', f) \end{cases}$$

These rules are illustrated in FIG. 5.9.

5.4.2 Composition of Relations

When several relations use the same activity as *right* part, we need to compose the *left* activities to obtain a consistent formula. The goal of this section is to formalize the intuitive semantics associated to the graphical representation presented above. The pressed reader may skip this formalization step and rely on his/her intuition. We consider here an activity $a \in \mathcal{A}$, involved in a process $p \in \mathcal{P}, a \in acts(p)$. To lighten the formula description, we use the p notation to refer to such a business process. Before describing in details the different functions used to compute such a formula, we provide here an intuitive description of these functions. For a given activity a , the following situations must be handled while computing $\varphi(a)$:

- Φ_w : *weak* predecessors reify an explicit disjunction in the entering flow,
- Φ_f : an exceptional-path reaches the control-path from which it diverges, and then implicitly defines a disjunction. Nested fault are handled by a recursive function Φ_d , dedicated to the identification of such disjunction,
- Φ_e : predecessors guarded by exclusive conditions implicitly define a disjunction,
- Φ_c : all others situations need to the conjunction of a' predecessors. A dedicated function Φ_b reify the way formulas are computed for a single (binary) relation.

² $end(a) \equiv$ “the activity a is in state end ”

³ $fail(a, f) \equiv$ “the activity a is in state $fail(f)$ ”

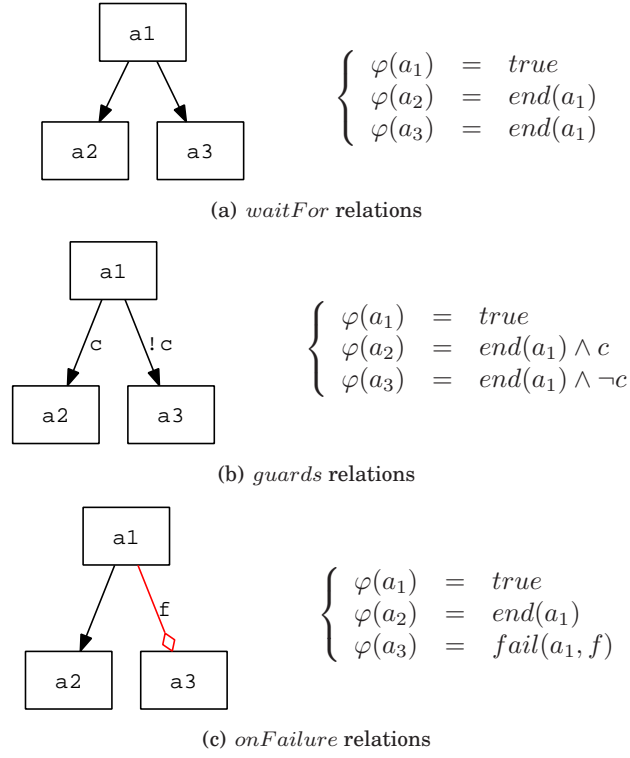


Figure 5.9: Trigger formulas associated to single-preceded activities

Composed Formula (Φ). For a given activity a , we use this function to build the final formula $\varphi(a)$ associated to a . It consists of a logical formula computed on the preceding activities (*preds* function) by the Φ_w function.

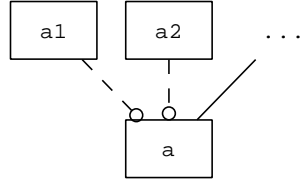
$$\begin{aligned} \Phi : \mathcal{A} &\rightarrow \text{Formula} \\ a &\mapsto \varphi(a) = \Phi_w(a, \text{preds}(a)) \\ \text{preds} : \mathcal{A} &\rightarrow \mathcal{A}^* \\ a &\mapsto \{a' \mid \exists a' < a \in \text{rels}(p)\} \end{aligned}$$

Handling Weak Relations (Φ_w). Using a *weakWait* relation, one defines a disjunction in the predecessor set. According to this semantics, we compute the formula as the disjunction of such predecessors and a formula expressed on the rest of the predecessors (Φ_f , see next paragraph). This function and the previous one are illustrated in FIG. 5.10.

$$\begin{aligned} \Phi_w : \mathcal{A} \times \mathcal{A}^* &\rightarrow \text{Formula} \\ (a, P) &\mapsto \left(\bigvee_{a' \in \text{weak}(a)} \text{end}(a') \right) \wedge \Phi_f(a, P \setminus \text{weak}(a)) \\ \text{weak} : \mathcal{A} &\rightarrow \mathcal{A}^* \\ a &\mapsto \{a' \mid \exists a' \ll a \in \text{rels}(p)\} \end{aligned}$$

Handling Fault Branches (Φ_f). Predecessors of the activity can be defined as children of a “*failure-fork*”, that is, a point where the control-path and an exceptional path diverge. We call such a point an *origin of failure*⁴ (the top-most point is computed through the *origin_f* function). Formulas associated to each branch (one per *origin of failure*, where such a “*failure-driven*”

⁴FIG. 5.11 defines two *origin of failure* for the predecessor set $\{a, b, c, d, e, f, g\}$: o_1 and o_3 (since o_2 is surrounded by o_1 , it is not considered at this step).



$$\begin{aligned}
\Phi(a) : \\
\Phi(a) &= \Phi_w(a, \text{preds}(a)) \\
\text{preds}(a) &= \{a_1, a_2, \dots\} \\
\Phi_w(a, \{a_1, a_2, \dots\}) : \\
P &= \{a_1, a_2, \dots\} \\
\text{weak}(a) &= \{a_1, a_2\} \\
\Phi_w(a, P) &= (\text{end}(a_1) \vee \text{end}(a_2)) \wedge \Phi_f(a, \{\dots\}) \\
\Rightarrow \varphi(a) &= (\text{end}(a_1) \vee \text{end}(a_2)) \wedge \Phi_f(a, \{\dots\})
\end{aligned}$$

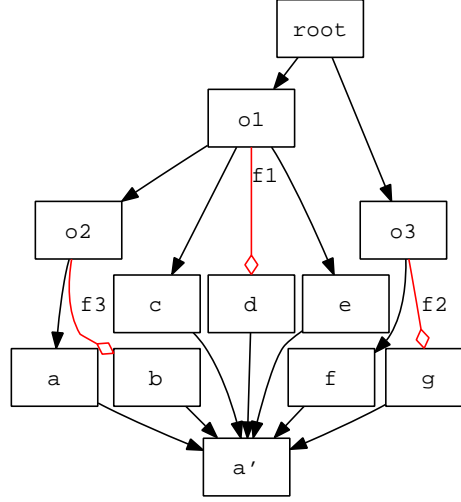
Figure 5.10: Building $\varphi(a)$: Φ and Φ_w illustration

partition is obtained through the Π_f function) are conjuncted together, and then conjuncted with the formula associated to predecessors without an adequate *origin of failure* in their ancestors. This function is illustrated in FIG. 5.11.

$$\begin{aligned}
\Phi_f : \mathcal{A} \times \mathcal{A}^* &\rightarrow \text{Formula} \\
(a, P) &\mapsto \left(\bigwedge_{b \in \Pi_f(a, P)} \Phi_d(a, b) \right) \wedge \Phi_d(a, P \setminus \bigcup (\Pi_f(a, P))) \\
\Pi_f : \mathcal{A} \times \mathcal{A}^* &\rightarrow (\mathcal{A}^*)^* \\
(a, P) &\mapsto \{b \mid \exists o \in \text{origin}_f^+(P), b = \text{branch}(o, a)\} \\
\text{origin}_f^+ : \mathcal{A}^* &\rightarrow \mathcal{A}^* \\
P &\mapsto \{o \mid o = \text{origin}_f(P)\} \\
\text{origin}_f : \mathcal{A}^* &\rightarrow \mathcal{A} \\
P &\mapsto o, \exists o \xrightarrow{f} x \in \text{rels}(p), \exists \alpha \in P, x \rightarrow \alpha \\
&\quad \wedge \nexists o' \in \text{acts}(p), o' \rightarrow o, o' \neq o, \exists \alpha' \in P, o' \xrightarrow{f} \alpha' \\
\text{branch} : \mathcal{A} \times \mathcal{A} &\rightarrow \mathcal{A}^* \\
(o, a) &\mapsto \{a' \mid \exists a' < a \in \text{rels}(p), o \rightarrow a'\}
\end{aligned}$$

Handling Disjunctive Flows (Φ_d). Considering a restricted subset of predecessors P (e.g., computed as a *branch* by Φ_f), we need to identify the control-path (thanks to the function f_c) and the different exceptional-paths (see FIG. 5.12) existing in P ancestors (such a partition is obtained through the Π_e function). For each subset of activities, we inductively call Φ_f to identify sub-branches and then associated formulas are built by induction. The stop-condition associated to this induction system is the reception of a subset of activities which only defines a control-path (i.e., there is no existing sub-branch defined inside this subset).

$$\begin{aligned}
\Phi_d : \mathcal{A} \times \mathcal{A}^* &\rightarrow \text{Formula} \\
(a, P) &\mapsto \begin{cases} f_c(a, P) = P \Rightarrow \Phi_e(a, P) \\ f_c(a, P) \neq P \Rightarrow \Phi_f(a, f_c(a, P)) \vee \left(\bigvee_{p \in \Pi_e(a, P)} \Phi_e(a, p) \right) \end{cases} \\
f_c : \mathcal{A} \times \mathcal{A}^* &\rightarrow \mathcal{A}^* \\
(a, P) &\mapsto \{a' \mid a' \in P, \exists o \in \text{origin}_f(a, P), o \xrightarrow{f} a'\}
\end{aligned}$$



$$\varphi(a') = \Phi(a') = \Phi_f(a', \{a, b, c, d, e, f, g\}) \quad (weaks(a') = \emptyset)$$

$$\Phi_f(a', \{a, b, c, d, e, f, g\}) :$$

$$\begin{aligned} \Pi_f(a', \{a, b, c, d, e, f, g\}) &= \{\{a, b, c, d, e\}, \{f, g\}\} \\ \text{result} &\rightsquigarrow \Phi_d(a', \{a, b, c, d, e\}) \wedge \Phi_d(a', \{f, g\}) \end{aligned}$$

$$\Phi_d(a', \{a, b, c, d, e\})$$

$$\begin{aligned} f_c(a', \{a, b, c, d, e\}) &= \{a, b, c, e\} \quad (\neq \{a, b, c, d, e\}) \\ \Pi_e(a', \{a, b, c, d, e\}) &= \{\{d\}\} \\ \text{result} &\rightsquigarrow \Phi_f(a', \{a, b, c, e\}) \vee \Phi_e(a', \{d\}) \end{aligned}$$

$$\Phi_f(a', \{a, b, c, e\}) :$$

$$\begin{aligned} \Pi_f(a', \{a, b, c, e\}) &= \{\{a, b\}\} \\ \text{result} &\rightsquigarrow \Phi_d(a, \{a, b\}) \wedge \Phi_d(a', \{c, e\}) \end{aligned}$$

$$\Phi_d(a', \{a, b\}) :$$

$$\begin{aligned} f_c(a', \{a, b\}) &= \{a\} \quad (\neq \{a, b\}) \\ \Pi_e(a', \{a, b\}) &= \{\{b\}\} \\ \text{result} &\rightsquigarrow \Phi_f(a', \{a\}) \vee \Phi_e(a', \{b\}) \\ &\rightsquigarrow \text{end}(a) \vee \text{end}(b) \end{aligned}$$

$$\Phi_d(a', \{c, e\}) :$$

$$\begin{aligned} f_c(a', \{c, e\}) &= \{c, e\} \quad (= \{c, e\}) \\ \text{result} &\rightsquigarrow \Phi_c(a', \{c, e\}) = \dots = \text{end}(c) \wedge \text{end}(e) \end{aligned}$$

$$\Phi_d(a', \{f, g\}) = \dots = \text{end}(f) \vee \text{end}(g)$$

$$\Rightarrow \varphi(a) = ((\text{end}(a) \vee \text{end}(b)) \wedge \text{end}(c) \wedge \text{end}(e)) \vee \text{end}(d) \wedge (\text{end}(f) \vee \text{end}(g))$$

Figure 5.11: building $\varphi(a')$: Φ_f and Φ_d inductive system

$$\begin{aligned}
\Pi_e : \mathcal{A} \times \mathcal{A}^* &\rightarrow (\mathcal{A}^*)^* \\
(a, P) &\mapsto \{flow \mid \exists o \in origin_f^+(P), \exists f \in faults(o), flow = f_e(a, P, o, f)\} \\
faults : \mathcal{A} &\rightarrow Term^* \\
a &\mapsto \{f \mid \exists a \xrightarrow{f} \alpha \in rels(p)\} \\
f_e : \mathcal{A} \times \mathcal{A}^* \times \mathcal{A} \times Term &\mapsto \mathcal{A}^* \\
(a, P, o, f) &\mapsto \{a' \mid a' \in P, \exists o \xrightarrow{f} \alpha \in rels(p), \alpha \rightarrow a'\}
\end{aligned}$$

Handling Exclusive Activities (Φ_e). We are considering now P a subset of a predecessors which does not contain any exclusivity induced by fault-catch mechanisms (inductively solved). Activities can be exclusive according to their transitive guards. Based on all the available guards expressed on P members (these *conditions* are obtained through the *conds* function), we compute the formula by using a dedicated function Φ_χ .

$$\begin{aligned}
\Phi_e : \mathcal{A} \times \mathcal{A}^* &\rightarrow Formula \\
(a, P) &\mapsto \Phi_\chi(A, P, conds(P)) \\
conds : \mathcal{A}^* &\rightarrow (\mathcal{V}^*)^* \\
P &\mapsto \{V \mid \forall a \in P, V = guards^+(a)\} \\
guards^+ \mathcal{A} &\rightarrow \mathcal{V}^* \\
a &\mapsto \{v \mid guard(a, v)\} \\
guard : \mathcal{A} \times \mathcal{V} &\rightarrow \mathbb{B} \\
(a, v) &\mapsto \exists g \xrightarrow{v} \alpha \in rels(p), \alpha \rightarrow a, \nexists g \xrightarrow{v} \alpha' \in rels(p), \alpha' \rightarrow a \\
&\quad \vee \exists g \xrightarrow{v} \alpha \in rels(p), \alpha \rightarrow a, \nexists g \xrightarrow{v} \alpha' \in rels(p), \alpha' \rightarrow a
\end{aligned}$$

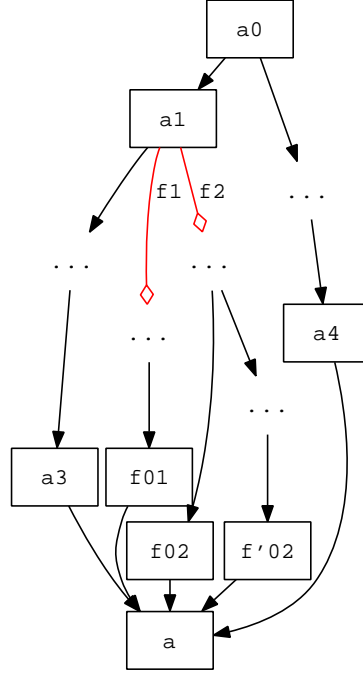
If the given set of conditions C is empty, there is no possible exclusivity between activities, and the computation of the formula is delegated to the Φ_c function which builds a conjunction of waits. In the other case, we start by picking *one*⁵ condition c in C , and *remove*⁶ it from C . We compute a partition of the predecessors set, according to their relation with the c condition: (i) predecessors guarded by $c = true$ (thanks to the *onTrue* function), (ii) predecessors guarded by $c = false$ (thanks to the *onFalse* function) and finally (iii) predecessors with no relation with c (function *notOn*).

The formula builds on this partition is by essence valid according to PROP. 4.10 (guard vivacity). We built the final formula thanks to a recursive call on Φ_χ , as the disjunction of formulas associated to *onTrue* and *onFalse* activities, conjuncted with the formula associated to the rest of the predecessors. An illustration of this function usage is depicted in FIG. 5.13.

$$\begin{aligned}
\Phi_\chi : \mathcal{A} \times \mathcal{A}^* \times (\mathcal{V}^*)^* &\rightarrow Formula \\
(a, P, C) &\mapsto \begin{cases} C = \emptyset &\Rightarrow \Phi_c(a, P) \\ C \neq \emptyset &\Rightarrow F \end{cases} \\
&\text{Let } c = one(C), C' = remove(c, C), \\
&F = \Phi_\chi(a, notOn(P, c), C') \wedge (\Phi_\chi(a, onTrue(P, c), C') \\
&\quad \vee \Phi_\chi(a, onFalse(P, c), C')) \\
onTrue : \mathcal{A}^* \times \mathcal{V} &\rightarrow \mathcal{A}^* \\
(P, v) &\mapsto \{a \mid \exists g \xrightarrow{v} \alpha \in rels(p), \alpha \rightarrow a, \nexists g \xrightarrow{v} \alpha' \in rels(p), \alpha' \rightarrow a\} \\
onFalse : \mathcal{A}^* \times \mathcal{V} &\rightarrow \mathcal{A}^* \\
(P, v) &\mapsto \{a \mid \exists g \xrightarrow{v} \alpha \in rels(p), \alpha \rightarrow a, \nexists g \xrightarrow{v} \alpha' \in rels(p), \alpha' \rightarrow a\} \\
notOn : \mathcal{A}^* \times \mathcal{V} &\rightarrow \mathcal{A}^* \\
(P, v) &\mapsto P \setminus (onTrue(P, v) \cup onFalse(P, v))
\end{aligned}$$

⁵We consider a function *one* : $(X^*)^* \rightarrow X$ which picks an element in the set which has the minimum cardinality. For example, *one*($\{\{a, b\}, \{a, b, c, d\}\}$) = a

⁶We consider a function *remove* : $X \times (X^*)^* \rightarrow (X^*)^*$ which removes an element in all the existing subsets. For example, *remove*($a, \{\{a, b\}, \{a, b, c, d\}\}$) = $\{\{b\}, \{b, c, d\}\}$



$\Phi(a) :$

$$\begin{aligned} \text{preds}(a) &= \{a_3, f_{01}, f_{02}, f'_{02}, a_4\} = P \\ \text{result} &\rightsquigarrow \Phi_w(a, P) = \Phi_f(a, P) \quad (\text{weak}(a) = \emptyset) \end{aligned}$$

$\Phi_f(a, P) :$

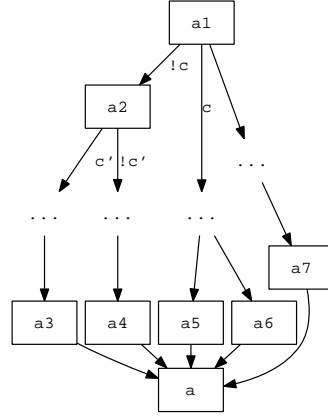
$$\begin{aligned} \Pi_f(a, P) &= \{\{a_3, f_{01}, f_{02}, f'_{02}\}\} \\ \text{result} &\rightsquigarrow \Phi_d(a, \{a_3, f_{01}, f_{02}, f'_{02}\}) \wedge \underbrace{\Phi_d(a, \{a_4\})}_{=end(a_4)} \end{aligned}$$

$\Phi_d(a, \{a_3, f_{01}, f_{02}, f'_{02}\}) :$

$$\begin{aligned} P' &= \{a_3, f_{01}, f_{02}, f'_{02}\} \\ f_c(a, P') &= \{a_3\} \quad (\neq P') \\ \Pi_e(a, P') &= \{\{f_{01}\}, \{f_{02}, f'_{02}\}\} \\ \text{result} &\rightsquigarrow \underbrace{\Phi_e(a, \{f_{01}\})}_{=end(f_{01})} \vee \underbrace{\Phi_e(a, \{f_{02}, f'_{02}\})}_{=end(f_{02}) \wedge end(f'_{02})} \vee \underbrace{\Phi_f(a, \{a_3\})}_{=end(a_3)} \end{aligned}$$

$$\Rightarrow \varphi(a) = (end(f_{01}) \vee (end(f_{02}) \wedge end(f'_{02})) \vee end(a_3)) \wedge end(a_4)$$

Figure 5.12: Building $\varphi(a)$: Multiple Faults handling in Φ_d



$\Phi(a) :$

$$\begin{aligned} \text{preds}(a) &= \{a_3, a_4, a_5, a_6, a_7\} = P \\ \text{result} &\rightsquigarrow \Phi_w(a, P) = \Phi_f(a, P) = \Phi_d(a, P) = \Phi_e(a, P) \end{aligned}$$

$\Phi_e(a, P) :$

$$\begin{aligned} \text{conds}(P) &= \{\{c\}, \{c, c'\}\} = \text{Conds} \\ \text{result} &\rightsquigarrow \Phi_\chi(a, P, \text{Conds}) \end{aligned}$$

$\Phi_\chi(a, P, \text{Conds}) :$

$$\begin{aligned} \text{one}(\text{Conds}) &= c, \quad \text{remove}(c, \text{Conds}) = \{\{c'\}\} \\ \text{onTrue}(P, c) &= \{a_5, a_6\}, \quad \text{onFalse}(P, c) = \{a_3, a_4\} \\ \text{notOn}(P, c) &= \{a_7\} \\ \text{result} &\rightsquigarrow (\Phi_\chi(a, \{a_5, a_6\}, \{\{c'\}\}) \vee \Phi_\chi(a, \{a_3, a_4\}, \{\{c'\}\})) \\ &\quad \wedge \Phi_\chi(a, \{a_7\}, \{\{c'\}\}) \end{aligned}$$

$\Phi_\chi(a, \{a_5, a_6\}, \{\{c'\}\}) :$

$$\begin{aligned} \text{one}(\{\{c'\}\}) &= c', \quad \text{remove}(c, \text{Conds}) = \emptyset \\ \text{onTrue}(\{a_5, a_6\}, c') &= \emptyset = \text{onFalse}(\{a_5, a_6\}, c') \\ \text{notOn}(\{a_5, a_6\}, c') &= \{a_5, a_6\} \\ \text{result} &\rightsquigarrow \Phi_\chi(a, \{a_5, a_6\}, \emptyset) = \underbrace{\Phi_c(a, \{a_5, a_6\})}_{\text{end}(a_5) \wedge \text{end}(a_6)} \end{aligned}$$

$\Phi_\chi(a, \{a_3, a_4\}, \{\{c'\}\}) :$

$$\begin{aligned} \text{onTrue}(\{a_3, a_4\}, c') &= \{a_3\}, \quad \text{onFalse}(\{a_3, a_4\}, c') = \{a_4\} \\ \text{notOn}(\{a_3, a_4\}, c') &= \emptyset \\ \text{result} &\rightsquigarrow \Phi_\chi(a, \{a_3\}, \emptyset) \vee \Phi_\chi(a, \{a_4\}, \emptyset) \\ &\rightsquigarrow \underbrace{\Phi_c(a, \{a_3\})}_{\text{end}(a_3)} \vee \underbrace{\Phi_c(a, \{a_4\})}_{\text{end}(a_4)} \end{aligned}$$

$\Phi_\chi(a, \{a_7\}, \{\{c'\}\}) :$

$$\begin{aligned} \text{onTrue}(\{a_7\}, c') &= \emptyset = \text{onFalse}(\{a_7\}, c') \\ \text{notOn}(\{a_7\}, c') &= \{a_7\} \\ \text{result} &\rightsquigarrow \Phi_\chi(a, \{a_7\}, \emptyset) = \underbrace{\Phi_c(a, \{a_7\})}_{\text{end}(a_7)} \end{aligned}$$

$$\Rightarrow \varphi(a) = ((\text{end}(a_3) \vee \text{end}(a_4)) \vee (\text{end}(a_5) \wedge \text{end}(a_6))) \wedge \text{end}(a_7)$$

Figure 5.13: Building $\varphi(a)$: Φ_e Illustration

Handling Conjunctive Flows (Φ_c). Considering a subset P of a predecessors without any exclusive activities, the formula associated to P is a conjunction of all the waits induced by the binary relation existence (computed through Φ_b).

$$\begin{aligned}\Phi_c : \mathcal{A} \times \mathcal{A}^* &\rightarrow Formula \\ (a, P) &\mapsto \bigwedge_{a' \in P} \Phi_b(a, a')\end{aligned}$$

Handling Binary Relation (Φ_b). According to the previous functions, we are now handling binary relations, and expect to transform it into a *Formula*. The result depends on the kind of relation defined in the relation *label*, as explained in SEC. 5.4.1.

$$\begin{aligned}\Phi_b : \mathcal{A} \times \mathcal{A} &\rightarrow Formula \\ (a, a') &\mapsto \begin{cases} a' \prec a \in rels(p) \Rightarrow end(a') \\ a' \stackrel{c}{\prec} a \in rels(p) \Rightarrow end(a') \wedge c \\ a' \stackrel{\neg c}{\prec} a \in rels(p) \Rightarrow end(a') \wedge \neg c \\ a' \stackrel{f}{\blacktriangleleft} a \in rels(p) \Rightarrow fail(a', f) \end{cases}\end{aligned}$$

5.5 Iteration Policy Handling

We handle predecessors associated to an iteration policy I_p (defined on data-set $d^* = \{d_1, \dots, d_n\}$) according to the *kind* of I_p . We base the semantics of these policies on the BPEL definition of the **forEach** composite activity:

“The **<forEach>** activity will execute its contained **<scope>** activity exactly **N+1** times where **N** equals the **<finalCounterValue>** minus the **<startCounterValue>**.

- If the value of the **parallel** attribute is **no** then the activity is a **serial <forEach>**. The enclosed **<scope>** activity **MUST** be executed **N+1** times, each instance starting only after the previous repetition is complete. [...]
- If the value of the **parallel** attribute is **yes** then the activity is a **parallel <forEach>**. The enclosed **<scope>** activity **MUST** be concurrently executed **N+1** times. [...]

[...] The **<forEach>** activity without a **<completionCondition>** completes when all of its child **<scope>**’s have completed. [...]

[OASIS, 2007]

Let I_p an iteration policy defined on d^* . According to this definition, we express the following mapping between ADORE and BPEL concepts:

$$\begin{aligned}\mathbf{forEach} &\equiv I_p \\ \mathbf{startCounterValue} &\equiv 1 \\ \mathbf{finalCounterValue} &\equiv |d^*| \\ \mathbf{scope} &\equiv acts(I_p) \\ \mathbf{parallel} &= \begin{cases} \mathbf{yes} &\equiv kind(I_p) = parallel \\ \mathbf{no} &\equiv kind(I_p) = serial \end{cases} \\ \mathbf{completionCondition} &\equiv \emptyset\end{aligned}$$

We denote as $end_i(a)$ the fact that the automaton associated to the data d_i for the activity a reaches its *end* state. The Φ^i algorithm is defined as an enhancement of the previously described algorithm which propagates the i concern in the computed formula for predecessors involved in an iteration policy. The behavioral distinction between *serial* and *parallel* policies can be defined in ADORE as the following:

- **Serial Policy:** The block contents is executed one by one, for each data. An illustration of this semantics is depicted in FIG. 5.15. It starts on the first data d_1 , using the previously defined semantics to trigger the first activity (S_2). Until the last data d_n , reaching the end

of the block for a data d_i means triggering the d_{i+1} execution (S_2). The last data d_n triggers the execution of the block successors (S_3). The other activities use the previously defined semantics (S_1).

- **Parallel Policy:** The block contents is concurrently executed for each $d_i \in d^*$. An illustration of this semantics is depicted in FIG. 5.16. It starts by triggering all the first activities (P_2). Blocks successors wait for the satisfaction of the conjunction of all φ_i formulas for the last activity of the block (P_3). The other activities use the previously defined semantics (P_1).

Composition with the previously defined semantics. Let $a \in \mathcal{A}$ an activity. To compute the *complete* execution semantic associated to a , we need to pre-process the predecessor set before using the previously defined algorithms. For each iteration policy associated to a predecessor of a , we compute the associated formula with a function Φ_I ⁷. The previously defined algorithm Φ_w is then executed on the remaining predecessors. The final formula is built as a conjunction of these formulas. An illustration of this composition is depicted in FIG. 5.14.

Limitation. We do not handle error policies associated to iteration policies, and we consider that no fault can be thrown by activities defined in the activity block associated to a policy. This is coherent with PROP. 4.19 (“no cross-boundary *onFailure* relation”) and also with the semantic associated to error-handling defined by the BPEL:

“If premature termination occurs such as due to a fault, [...] then this **N+1** requirement does not apply.” [OASIS, 2007]

Considering iteration as equivalent to a final recursion, one can bypass this limitation, but must write the process using this style of writing (and rely on the execution engine to optimize the execution). Enhancing ADORE expressiveness to address in a better way error handling and error composition for iteration policies is exposed as one of the perspective of this work.

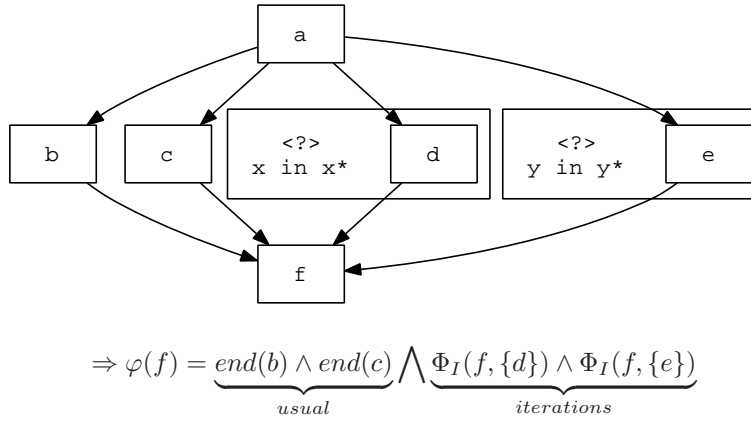
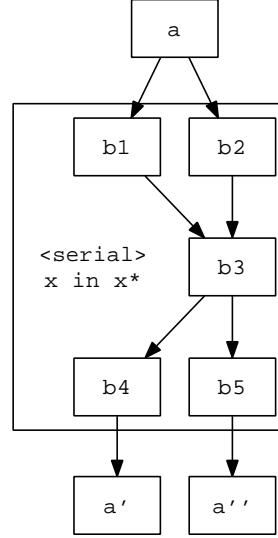


Figure 5.14: Composition of iteration and control-flow

⁷Defined as an implementation of the previously described mechanisms



DEFINITIONS :

$$\begin{aligned}
 d^* &= \{d_1, \dots, d_n\}, |d^*| = n \\
 B &= \{b_1, b_2, b_3, b_4, b_5\} \\
 firsts(B) &= \{b_1, b_2\} = F \\
 nexts(B) &= \{a', a''\} = N
 \end{aligned}$$

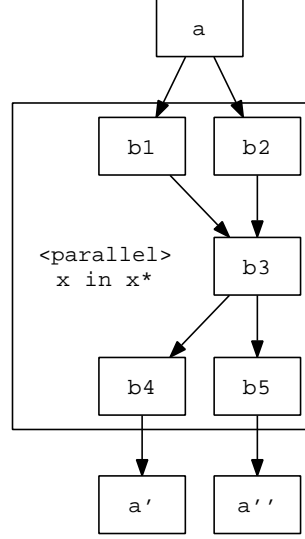
GENERAL DESCRIPTION :

$$\begin{aligned}
 (S_1) \quad \forall a \in B \setminus F, \quad \varphi_i(a) &= \Phi^i(a) \\
 (S_2) \quad \forall a \in F, \quad \varphi_1(a) &= \Phi(a) \\
 &\quad \varphi_{2 \leq i \leq n}(a) = \bigwedge_{\nu \in nexts} \Phi^{i-1}(\nu) \\
 (S_3) \quad \forall a \in N, \quad \varphi(a) &= \Phi^n(a)
 \end{aligned}$$

COMPUTED EXECUTION SEMANTICS :

$$\begin{aligned}
 \varphi(a) &= true \\
 \varphi_i(b_3) &= end_i(b_1) \wedge end_i(b_2) & (S_1) \\
 \varphi_i(b_4) &= end_i(b_3) & (S_1) \\
 \varphi_i(b_5) &= end_i(b_3) & (S_1) \\
 \varphi_1(b_1) &= end(a) & (S_2) \\
 \varphi_{2 \leq i \leq n}(b_1) &= end_{i-1}(b_4) \wedge end_{i-1}(b_5) & (S_2) \\
 \varphi_1(b_2) &= end(a) & (S_2) \\
 \varphi_{2 \leq i \leq n}(b_2) &= end_{i-1}(b_4) \wedge end_{i-1}(b_5) & (S_2) \\
 \varphi(a') &= end_n(b_4) & (S_3) \\
 \varphi(a'') &= end_n(b_5) & (S_3)
 \end{aligned}$$

Figure 5.15: Execution semantics associated to *serial* Iteration Policies



DEFINITIONS :

$$\begin{aligned}
 d^* &= \{d_1, \dots, d_n\}, |d^*| = n \\
 B &= \{b_1, b_2, b_3, b_4, b_5\} \\
 firsts(B) &= \{b_1, b_2\} = F \\
 nexts(B) &= \{a', a''\} = N
 \end{aligned}$$

GENERAL DESCRIPTION :

$$\begin{aligned}
 (P_1) \quad \forall a \in B \setminus F, \quad \varphi_i(a) &= \Phi^i(a) \\
 (P_2) \quad \forall a \in F, \quad \varphi_i(a) &= \Phi(a) \\
 (P_3) \quad \forall a \in N, \quad \varphi(a) &= \bigwedge_{i=1 \dots n} \Phi^n(a)
 \end{aligned}$$

COMPUTED EXECUTION SEMANTICS :

$$\begin{aligned}
 \varphi(a) &= true \\
 \varphi_i(b_3) &= end_i(b_1) \wedge end_i(b_2) & (P_1) \\
 \varphi_i(b_4) &= end_i(b_3) & (P_1) \\
 \varphi_i(b_5) &= end_i(b_3) & (P_1) \\
 \varphi_i(b_1) &= end(a) & (P_2) \\
 \varphi_i(b_2) &= end(a) & (P_2) \\
 \varphi(a') &= \bigwedge_{i=1 \dots n} end_n(b_4) & (P_3) \\
 \varphi(a'') &= \bigwedge_{i=1 \dots n} end_n(b_5) & (P_3)
 \end{aligned}$$

Figure 5.16: Execution semantics associated to *parallel* Iteration Policies

Interacting with ADORE: Actions & Algorithms

«Great things are done by a series of small things brought together.»
Vincent van Gogh

Contents

6.1 Introduction	69
6.2 Logical Foundations	70
6.2.1 Equivalence (\equiv)	70
6.2.2 Substitutions (θ)	70
6.3 The ADORE Action Language	71
6.3.1 Elementary Actions: Add & Del	71
6.3.2 Action Lists, Execution & Model Consistency	73
6.3.3 Composite Actions: Replace, Unify & Discharge	74
6.4 Defining Algorithms to Interact with Models	77
6.4.1 Traceability: Algorithm Context & Symbol Generation	77
6.4.2 Illustration #1: A Cloning Algorithm	78
6.4.3 Illustration #2: Naive Process Correlation	78
6.4.4 Composition Algorithms Defined using ADORE	78

6.1 Introduction

The ADORE meta-model aims to compose business processes together, following the *Separation of Concerns* paradigm. As a consequence, we need to formally define mechanisms to interact with ADORE models. State-of-art approaches such as the MOF reflexive API [OMG, 2006] use an action language to perform such a goal. The D-PRAXIS approach advocates an action-based approach of model representation:

“Every model can be expressed as a sequence of elementary construction operations. The sequence of operations that produces a model is composed of the operations performed to define each model element.”

[Blanc et al., 2008]

In this chapter, we define in SEC. 6.2 the logical foundations underlying the approach. Then, we describe in SEC. 6.3 the different actions defined over the ADORE meta-model. Finally, SEC. 6.4 shows how these actions can be used to define *algorithms*.

6.2 Logical Foundations

In this section, we enhance the ADORE meta-model to equip it with an equivalence relation (used to *match* equivalent elements while composing elements). We also explain how the *logical substitution* mechanism can be used to perform model derivation¹ in a simple way.

6.2.1 Equivalence (\equiv)

In the previous chapters, we use the equality symbol $=$ in its usual form, that is, two entities defined in ADORE are equals since they are exactly the same. This *plain equality* relation is too strong when talking about composition algorithms, since such algorithms identify *equivalent* entities² and then work on it. An informal definition of the ADORE equivalence relation (denoted as \equiv) can be the following: “*two entities are equivalent since their contents are equivalent, excepting their names*”. The equivalence of closed terms is assimilated as their equality. When talking about sets, we consider two sets as equivalent since they are equipollent according to the \equiv relation.

$$\begin{aligned}
\equiv: \quad \mathcal{V} \times \mathcal{V} &\rightarrow \mathbb{B} \\
(v, v') &\mapsto \text{type}(v) = \text{type}(v') \wedge \text{isSet?}(v) = \text{isSet?}(v') \\
\equiv: \quad \mathcal{A} \times \mathcal{A} &\rightarrow \mathbb{B} \\
(a, a') &\mapsto \text{kind}(a) = \text{kind}(a') \wedge \text{inputs}(a) \equiv \text{inputs}(a') \wedge \text{outputs}(a) \equiv \text{outputs}(a') \\
\equiv: \quad \mathcal{R} \times \mathcal{R} &\rightarrow \mathbb{B} \\
(r, r') &\mapsto \text{left}(r) \equiv \text{left}(r') \wedge \text{right}(r) \equiv \text{right}(r') \wedge \text{label}(r) = \text{label}(r') \\
\equiv: \quad \mathcal{P} \times \mathcal{P} &\times \mathbb{B} \\
(p, p') &\mapsto \text{vars}(p) \equiv \text{vars}(p') \wedge \text{acts}(p) \equiv \text{acts}(p') \wedge \text{rels}(p) \equiv \text{rels}(p')
\end{aligned}$$

6.2.2 Substitutions (θ)

Composition algorithms often create new entities which derive from the legacy ones³. According to the logical background of ADORE, we use *logical substitutions* [Stickel, 1981] to allow one to build new elements by modifying existing ones. The complete definition of *logical substitution* is given below:

Substitution Component. “A substitution component is an ordered pair of a variable v and a term t written as $v \leftarrow t$. A substitution component denotes the assignment of the term to the variable or the replacement of the variable by the term.”

Substitution. “A substitution is a set of substitution components with distinct first elements, that is, distinct variables being substituted for. Applying a substitution to an expression results in the replacement of those variables of the expression included among the first elements of the substitution components by the corresponding terms. The substitution components are applied to the expression in parallel, and no variable occurrence in the second element of a substitution component is replaced even if the variable occurs as the first element in another substitution component. Substitutions are represented by the symbols σ , θ , and ϕ . The application of substitution θ to expression A is denoted by $A\theta$. The composition of substitutions $\theta\sigma$ denotes the substitution whose effect is the same as first applying substitution θ , then applying substitution σ ; that is, $A(\theta\sigma) = (A\theta)\sigma$ for every expression A .” [Stickel, 1981]

¹We use the word “derivation” according to its linguistic definition: “**derivation**: Used to form new words, as with *happi-ness* and *un-happy* from *happy*, or *determination* from *determine*.” [Crystal, 1999]

²ADORE entities are only structural, and passed by values according to the BPEL semantics.

³e.g., in the KOMPOSE framework: “*Merging: **matched model elements** are merged to create **new model elements** that represent an integrated view of the concepts.*” [Fleurey et al., 2007]

Example. Let a an activity, and v, v' two variables. The activity a uses the variable v . Using two substitution components $v \leftarrow v'$ and $a \leftarrow a'$, one can create an activity a' which uses the variable v' instead of v (i.e., uses a **float** instead of a **double**) by applying a substitution.

$$\begin{aligned} v &= (v, \text{double}, \text{false}) \in \mathcal{V}, & v' &= (v', \text{float}, \text{false}) \in \mathcal{V} \\ a &= (a, \text{receive}, \emptyset, \{v\}) \in \mathcal{A}, & \theta &= \{a \leftarrow a', v \leftarrow v'\} \\ a\theta &= (a', \text{receive}, \emptyset, \{v'\}) \in \mathcal{A} \end{aligned}$$

6.3 The ADORE Action Language

The MetaObject Facility (MOF [OMG, 2006]) defines⁴ a reflexive API used to manipulate model elements. This API, dedicated to the class-based MOF meta-model, describes seven atomic operations used to handle model-element through reflection. The PRAXIS method [Blanc et al., 2008] restricts this set of operations to four, allowing one to (i) **create** a model element, (ii) **delete** a model element, (iii) set a property in a model element (**setProperty**) and finally (iv) set a reference from a model element to another one (**setReference**). Each operation defined in PRAXIS exhibits a *precondition*. It is interesting to notice that the action-based representation supports naturally a collaborative development approach [Mougenot et al., 2009].

Considering ADORE as a meta-model dedicated to business processes, we decide to specialize the concepts expressed in the previous languages to build a *domain-specific* language dedicated to the handling of ADORE elements. In this context, we define two elementary actions (**add** & **del**), and three *composite* actions defined as *macros* over the elementary ones (**replace**, **unify** & **discharge**). Each ADORE action requires a precondition and ensures a postcondition. We decide to use a *functional* representation of the actions, considering that the execution of an action returns a **new** universe where the postcondition is true, instead of modifying the existing one through a side-effect. To lighten the description of the mechanisms, we refer to ADORE *elements* in the same way one refers to “*model elements*” when talking about class instances. We use the \mathcal{E} symbol to reify the **Element** concept, formally defined as the union of all existing concepts defined in ADORE. We denote as *Action* the set of all existing actions. To lighten the description of the logical formulas, we indifferently accept for an *element* the usage of the element itself or its identifier (its *name*).

$$\mathcal{E} = (\mathcal{V} \cup \mathcal{A} \cup \mathcal{R} \cup \mathcal{P} \cup \mathcal{U} \cup \mathcal{I})$$

Execution. An action is performed on a given universe $u \in \mathcal{U}$, where a *precondition* can be checked (modeled as a boolean function). We denote as *do* the function used to perform an action. Since its precondition is true, an action cannot fail and always returns an universe where its postcondition is true (modeled as a boolean function).

$$\begin{aligned} do : Action \times \mathcal{U} &\rightarrow \mathcal{U} \\ (\alpha, u) &\mapsto \begin{cases} \neg precondition(\alpha, u) \vee u = nil \Rightarrow nil \\ precondition(\alpha, u) \Rightarrow do(\alpha, u) = u' \wedge postcondition(\alpha, u') \end{cases} \end{aligned}$$

6.3.1 Elementary Actions: Add & Del

In this section, we describe the two elementary actions available on ADORE elements: *add* and *del*. Using these actions, one can change a *container* element (i.e., a business process or an universe) by adding (respectively removing) an element inside the container. To lighten the description, we assume that the *container* exists in the universe used to perform the action. Executing an elementary action only modifies the *container* element, *ceteris paribus*.

⁴Pages 27 → 30 of the specification, <http://www.omg.org/spec/MOF/2.0/PDF/>

Add. This operation is used to create an element and insert it into another one, *e.g.*, adding an activity into a business process. It takes as input a ground term (the kind of concept to be added, *e.g.*, *activity*), the *element* to be added and its future *container*. We use a term of arity three to represent it.

$$\text{add}(\text{kind}, \text{element}, \text{container}) \in \text{Action}, \text{kind} \in \text{GroundTerm}, \text{element} \in \mathcal{E}, \text{container} \in \mathcal{E}$$

According to the *kind* parameter, an *add* action will check structural preconditions. Since the semantic of an *add* is to augment the original model, executing such an action returns an universe which contains the original one. If an element with the same *name* still exists in the universe, the *add* action will silently retract it before performing the add. Thanks to this property, the *add* action is intrinsically idempotent.

$$\text{do}(\text{add}(k, e, c), u) = u' \Rightarrow \text{do}(\text{add}(k, e, c), \text{do}(\text{add}(k, e, c), u)) = u' \quad (\text{Idempotency})$$

We describe in TAB. 6.1 the preconditions and postconditions associated to the *add* action. We define a syntactic shortcut to lighten the usage of the *add* action: the *kind* is shortened to its first letter and then used as an index of the *add* symbol. Precondition (1) ensures the respect of PROP. 4.15 (process isolation) at the activity level. Precondition (2) handles PROP. 4.10 (guard vivacity) at the relation level. Finally, precondition (3) checks the intrinsic correctness of an iteration policy.

$\alpha = \text{add}(\text{kind}, \text{element}, \text{container})$				$u \in \mathcal{U}, \text{do}(\alpha, u)$	
	<i>kind</i>	<i>element</i>	<i>container</i>	Precondition	Postcondition
$\text{add}_v(v, p) \equiv$	<i>variable</i>	$v \in \mathcal{V}$	$p \in \mathcal{P}$	—	$v \in \text{vars}(p)$
$\text{add}_a(a, p) \equiv$	<i>activity</i>	$a \in \mathcal{A}$	$p \in \mathcal{P}$	(1)	$a \in \text{acts}(p)$
$\text{add}_r(r, p) \equiv$	<i>relations</i>	$r = (a < a') \in \mathcal{R}$	$p \in \mathcal{P}$	(2)	$r \in \text{rels}(p)$
$\text{add}_p(p) \equiv$	<i>process</i>	$p \in \mathcal{P}$	$u \in \mathcal{U}$	—	$p \in \text{procs}(u)$
$\text{add}_i(i, p) \equiv$	<i>iteration</i>	$i \in \mathcal{I}$	$p \in \mathcal{P}$	(3)	$i \in \text{pols}(p)$

- (1) $\equiv \forall v \in \text{vars}(a), v \in \text{vars}(p)$
(2) $\equiv a \in \text{acts}(p), a' \in \text{acts}(p), \nexists a' \rightarrow a$
 $\quad \wedge (\text{label}(r) = \text{guard}(v, b) \Rightarrow v \in \text{vars}(p) \wedge v \in \text{outputs}(a))$
(3) $\equiv \text{set}(i) \in \text{vars}(p) \wedge \text{scalar}(i) \in \text{vars}(p)$
 $\quad \wedge \forall a \in \text{acts}(i), (a \in \text{acts}(p) \wedge \text{set}(i) \notin \text{outputs}(a))$

Table 6.1: Preconditions & Postconditions associated to the *add* action.

Del. This operation is used to remove an *element* from its *container*, according to its *kind*. We use a term of arity three to represent it.

$$\text{del}(\text{kind}, \text{element}, \text{container}) \in \text{Action}, \text{kind} \in \text{GroundTerm}, \text{element} \in \mathcal{E}, \text{container} \in \mathcal{E}$$

The semantics of a *del* action is to remove an element in the original model, and the execution of such an action returns an universe which is contained by the original one. The general idea of the *del* preconditions is to allow the deletion of an element only when it is never “used” (*e.g.*, deleting a variable still used as input or output in an activity will be refused) in the container. The deletion of a non-existing element is accepted, and basically returns the original universe. As a consequence, the *del* action is idempotent. Deleting an element after adding it is equivalent to do nothing, and returns the same universe as the one existing before the *add*. Precondition (1) ensures that a variable cannot be deleted if an activity is still using it. Precondition (2) does the same at the activity level: an activity cannot be deleted if still related to another one.

$$\begin{aligned} \text{do}(\text{del}(k, e, c), u) = u' &\Rightarrow \text{do}(\text{del}(k, e, c), \text{do}(\text{del}(k, e, c), u)) = u' && (\text{Idempotency}) \\ u &= \text{do}(\text{del}(k, e, c), \text{do}(\text{add}(k, e, c), u)) && (\text{del} \circ \text{add composition}) \end{aligned}$$

We present in TAB. 6.2 the associated preconditions and postconditions. We use the same syntactic shortcut than the *add* one.

$\alpha = del(kind, element, container)$				$u \in \mathcal{U}, do(\alpha, u) = u'$	
	<i>kind</i>	<i>element</i>	<i>container</i>	Precondition	Postcondition
$del_v(v, p) \equiv$	<i>variable</i>	$v \in \mathcal{V}$	$p \in \mathcal{P}$	(1)	$v \notin vars(u')$
$del_a(a, p) \equiv$	<i>activity</i>	$a \in \mathcal{A}$	$p \in \mathcal{P}$	(2)	$a \notin acts(u')$
$del_r(r, p) \equiv$	<i>relations</i>	$r = (a < a') \in \mathcal{R}$	$p \in \mathcal{P}$	–	$r \notin rels(u')$
$del_p(p) \equiv$	<i>process</i>	$p \in \mathcal{P}$	$u \in \mathcal{U}$	–	$p \notin procs(u')$
$del_i(i, p) \equiv$	<i>iteration</i>	$i \in \mathcal{I}$	$p \in \mathcal{P}$	–	$i \notin polys(u')$

$$(1) \equiv \nexists a \in acts(p), v \in vars(a) \wedge \nexists (a \prec a') \in rels(p)$$

$$(2) \equiv \nexists r \in rels(p), (left(r) = a \vee right(r) = a)$$

Table 6.2: Preconditions & Postconditions associated to the *del* action.

6.3.2 Action Lists, Execution & Model Consistency

Considering an ordered list⁵ of actions $A = (\alpha_1, \dots, \alpha_n) \in Actions_{<}^*$, we define the execution of A on an universe $u \in \mathcal{U}$ as the *functional folding* [Iverson, 1962] of *do* (denoted as do^+).

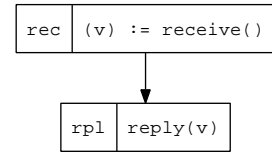
$$do^+ : Actions_{<}^* \times \mathcal{U} \rightarrow \mathcal{U}$$

$$(A, u) \mapsto \begin{cases} A = \emptyset & \Rightarrow u \\ A \neq \emptyset & \Rightarrow do^+(tail(A), do(head(A), u)) \end{cases}$$

Example. We consider here a simple orchestration of services, which receives an integer and immediately returns it. We describe in FIG. 6.1 a sequence of actions S which builds such a process.

$$\begin{aligned} \alpha_1 &= add_p((example, \emptyset, \emptyset, \emptyset)) \\ \alpha_2 &= add_v((v, integer, false), example) \\ \alpha_3 &= add_a((rec, receive, \emptyset, \{v\}), example) \\ \alpha_4 &= add_a((rpl, reply, \emptyset, \{v\}), example) \\ \alpha_5 &= add_r((rec, rpl, waitFor), example) \\ S &= (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) \end{aligned}$$

(a) S , a sequence of elementary actions



Orchestration simple::example

(b) Process obtained after the execution of S

Figure 6.1: A sequence of actions, and the result of its execution

Ensuring Model Consistency. We defined in SEC. 4.4 a set of properties which makes ADORE models *consistent*, and assumes that all these rules are respected. Even if the precondition associated to each action avoids several inconsistencies at a fine-grained level (e.g., type checking), we cannot ensure that the coarse-grained properties (e.g., “entry-to-exit” path existence, PROP. 4.12) are respected. Moreover, it is normal for a model to be inconsistent while creating it [Balzer, 1991].

⁵We consider a *list* as usually defined in the literature [Cormen et al., 2001], with function such as *head* to retrieve its first element, *tail* to access to the others and $\hat{\cdot}$ to append an element at its end. We use the $\mathcal{E}_{<}^*$ notation to denote a “list of \mathcal{E} ”

As a consequence, we consider that models handled *while executing an action list* can be inconsistent according to the ADORE constraints. But the final model obtained after the execution of the last action **must** be consistent. An inconsistent model will be rejected by the system at this step. Considering that each model handled in ADORE was created by executing an action list, we can inductively ensure that all models handled in the framework are consistent.

Such an issue is illustrated in the previous example. For simplification purpose, we focus on a single property: the “*entry-to-exit*” path existence (PROP. 4.12). Assuming that the initial universe $u \in \mathcal{U}$ is consistent, the computed universes retrieved after the execution of α_1 and α_2 are still consistent according to this property. But since the α_3 action is executed, the constraint is violated and the computed universe is inconsistent. The execution of α_5 fixes this inconsistency and returns a consistent model, accepted by the framework.

$$\underbrace{do^+(S, u) = u'}_{\text{consistent}} \equiv do(\alpha_5, \underbrace{do(\alpha_4, \overbrace{do(\alpha_3, do(\alpha_2, do(\alpha_1, u)))})}_{\text{inconsistent}}))$$

6.3.3 Composite Actions: Replace, Unify & Discharge

Since the previously described actions are elementary and technically-oriented, we define more complex actions by combining them. Such *composite* actions ease the description of composition mechanisms in next chapters. We define these actions as macros, that is, actions which can be refined into elementary ones. The execution semantic of these macro-actions is equivalent to the execution of the sequence of elementary actions associated to a macro, statically computed before starting the execution. As a consequence, preconditions and postconditions are checked for each executed elementary action.

Replace. The *replace* action allows one to change all usages of an element (named *old*) by another one (named *new*). The *replace* action is modeled as a term of arity four: the *kind* of replacement, the *old* element, the *new* one, and the container p . We define a short notation $r(old, new, p)$ which makes implicit the kind of replacement to lighten the descriptions. Executing a *replace* action means to add the *new* element, substitute all usage of *old* in p by a usage of *new*, and then delete the *old* element. In ADORE, we only allow the replacement of variables and activities, contained by a given process $p \in \mathcal{P}$. A replacement is valid only between elements from the same kind (*i.e.*, an activity cannot be replaced by a variable).

$$\begin{aligned} kind &\in \text{GroundTerm}, \text{ old } \in (\mathcal{V} \cup \mathcal{A}), \text{ new } \in (\mathcal{V} \cup \mathcal{A}), p \in \mathcal{P}, \\ r(old, new, p) &\equiv \text{replace}(kind, old, new, p) \in \text{Action} \end{aligned}$$

We describe in the following paragraph the elementary actions associated to each kind of replacement defined in ADORE.

$$r(v, v', p) \equiv \text{replace}(\text{variable}, v, v', p) \in \text{Action}, v \in \mathcal{V}, v' \in \mathcal{V}, p \in \mathcal{P} : \quad \theta = \{v \leftarrow v'\}$$

$$\text{propagate}_a = \{r(a, a\theta, p) \mid \exists a \in \text{acts}(p), v \in \text{vars}(a)\}$$

$$\text{propagate}_r = \{(del_r(r, p), add_r(r\theta, p)) \mid \exists r = a \stackrel{v}{\prec} a' \in \text{rels}(p)\}$$

$$\text{policies} = \{del_i(i, p), add(i\theta, p) \mid \exists i \in \text{pols}(p), v = \text{scalar}(i) \vee v = \text{set}(i)\}$$

$$\rightsquigarrow (add_v(v', p)) \stackrel{\pm}{\leftarrow} \text{propagate}_a \stackrel{\pm}{\leftarrow} \text{propagate}_r \stackrel{\pm}{\leftarrow} \text{policies} \stackrel{\pm}{\leftarrow} del_v(v, p)$$

$$r(a, a', p) \equiv \text{replace}(\text{activity}, a, a') \in \text{Action}, a \in \mathcal{A}, a' \in \mathcal{A}, p \in \mathcal{P} : \quad \theta = \{a \leftarrow a'\}$$

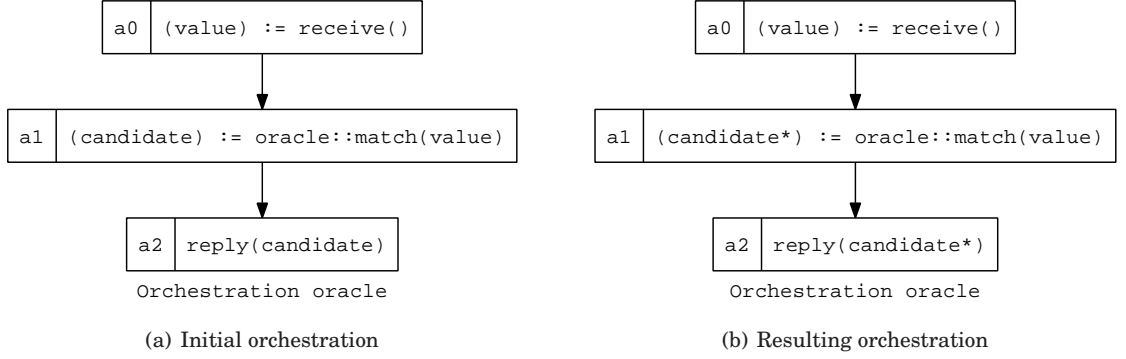
$$\text{clean} = \{del_r(a < x, p) \mid \exists a < x \in \text{rels}(p)\} \cup \{del_r(x < a, p) \mid \exists x < a \in \text{rels}(p)\}$$

$$\text{reorder} = \{add_r((a', x, k), p) \mid \exists (a, x, k) \in \text{rels}(p)\} \cup \{add_r((x, a', k), p) \mid \exists (x, a, k) \in \text{rels}(p)\}$$

$$\text{policies} = \{(del_i(i), add_i(i\theta, p)) \mid \exists i \in \text{pols}(p), a \in \text{acts}(i)\}$$

$$\rightsquigarrow () \stackrel{\pm}{\leftarrow} \text{clean} \stackrel{\pm}{\leftarrow} (del_a(a, p), add_a(a', p)) \stackrel{\pm}{\leftarrow} \text{reorder} \stackrel{\pm}{\leftarrow} \text{policies}$$

We illustrate the *replace* action using a simple process, in FIG. 6.2(a). This business process $oracle \in \mathcal{O}$ receives a *value* (a_0), asks an oracle service to find a matching *candidate* (a_1) and then replies such a data to the caller (a_2). Considering that the a_1 invocation can now return a set of candidates instead of a single one, we can use a *replace* action $r(candidate, candidate^*)$ to take care of this requirement. The sequence of elementary actions obtained as a refinement of this composite action is depicted in FIG. 6.2(c).



$$\begin{aligned}
 \theta &= \{candidate \leftarrow candidate^*\} \\
 \alpha_1 &= add_v((candidate^*, data, true), p) \\
 r(a_1, a_1\theta, oracle) &\rightsquigarrow \begin{cases} \alpha_2 = del_r(a_0 \prec a_1, oracle), \alpha_3 = del_r(a_1 \prec a_2, oracle) \\ \alpha_4 = del_a(a_1, oracle) \\ \alpha_5 = add_a((a_1, invoke(...), \{value\}, \{candidate^*\}), oracle) \\ \alpha_6 = add_r(a_0 \prec a_1, oracle), \alpha_7 = add_r(a_1 \prec a_2, oracle) \end{cases} \\
 r(a_2, a_2\theta, oracle) &\rightsquigarrow \begin{cases} \alpha_8 = del_r(a_1 \prec a_2, oracle), \alpha_9 = del_a(a_2, oracle) \\ \alpha_{10} = add_a((a_2, reply, \{candidate^*\}, \{\}), oracle) \\ \alpha_{11} = add_r(a_1 \prec a_2, oracle) \end{cases} \\
 S &= (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7, \alpha_8, \alpha_9, \alpha_{10}, \alpha_{11})
 \end{aligned}$$

(c) Sequence S of elementary actions equivalent to α

Figure 6.2: Illustrating the *replace* action: $\alpha = r(candidate, candidate^*, oracle)$

Unify. This action allows one to unify several model elements into a single one. It is modeled as a term of arity three, where the first parameter is the set of model elements $e^* = \{e_1, \dots, e_n\}$ to be unified and the second one is the expected *name* of the unified element. This action makes sense in the context of a given process $p \in \mathcal{P}$ which contains all the entities to be unified. We use the same notation than the one used for the *replace* action: a complete term of arity four, and a shortened version where the *kind* of unification is implicit.

$$\begin{aligned}
 e^* &\in (\mathcal{V}^* \cup \mathcal{A}^*), \text{ name} \in \text{GroundTerm}, p \in \mathcal{P} \\
 u(e^*, \text{name}, p) &\equiv \text{unify}(\text{kind}, e^*, \text{name}, p) \in \text{Action}
 \end{aligned}$$

We describe here the elementary actions associated to the unification of a variable set. We assume that unification candidates are coherent: a set of variables to be unified must define an equivalence class according to the \equiv relation. We use the *one* function previously defined in SEC. 5.4.2 to access to a given candidate contained in the candidate set.

$$u(V, v', p) \equiv \text{unify}(\text{variable}, V, v', p), \quad V = \{v_1, \dots, v_n\} \in \mathcal{V}^*, v' \in \text{GroundTerm}; p \in \mathcal{P} :$$

Precondition: $\forall (\nu, \nu') \in V^2, \nu \equiv \nu'$. Let $v = \text{one}(V) \in \mathcal{V}$,

$$\text{guards} = \{(del_r(a \prec a', p), add_r(a \prec a', p)) \mid \exists (a \prec a') \in \text{rels}(p), v \in V\}$$

$$\rightsquigarrow (add_v((v', type(v), isSet?(v)), p)) \stackrel{\perp}{\leftarrow} \{r(v, v', p) \mid \exists v \in V\} \stackrel{\perp}{\leftarrow} guards$$

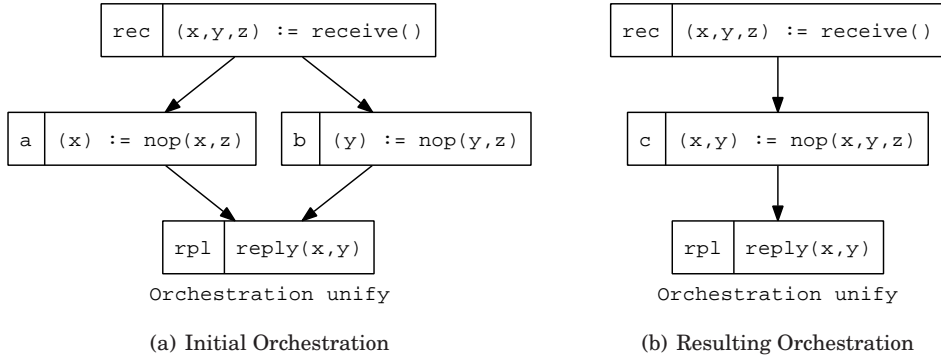
The unification of a set of activities $A \equiv \{a_1, \dots, a_n\}$ (holding the same *kind*) produces an activity a' defined through the union of inputs and outputs variables: $vars(a') = \bigcup_{a \in A} vars(a)$.

$$u(A, a', p) \equiv unify(activity, A, a', p), \quad A = \{a_1, \dots, a_n\} \in \mathcal{A}^*, a' \in GroundTerm, p \in \mathcal{P} :$$

Precondition: $\forall(\alpha, \alpha') \in A, kind(\alpha) = kind(\alpha')$

$$\rightsquigarrow (add_a((a', kind(one(A)), \bigcup_{a \in A} inputs(a), \bigcup_{a \in A} outputs(a)), p)) \stackrel{\perp}{\leftarrow} \{r(a, a', p) \mid \exists a \in A\}$$

We illustrate the *unify* action on the example depicted in FIG. 6.3. The original orchestration depicted in FIG. 6.3(a) contains four activities: $\{rec, a, b, rpl\}$. One can decide to *unify* the activities a and b to produce a new activity, named c . Such an action can be realized by using a *unify*($\{a, b\}, c$) action, which produces the associated elementary action sequence S (FIG. 6.3(c)). The process obtained after the execution of S is depicted in FIG. 6.3(b)



$$\begin{aligned}
 \alpha_1 &= add_a((c, nop, \{x, y, z\}, \{x, y\}), unify) \\
 r(a, c, unify) &\rightsquigarrow \begin{cases} \alpha_2 = del_r(rec \prec a, unify), \alpha_3 = del_r(a \prec rpl, unify) \\ \alpha_4 = del_a(a, unify) \\ \alpha_5 = add_a((c, nop, \{x, y, z\}, \{x, y\}), unify) \\ \alpha_6 = add_r(rec \prec c, unify) \\ \alpha_7 = add_r(c \prec rpl, unify) \end{cases} \quad (= \alpha_1) \\
 r(b, c, unify) &\rightsquigarrow \begin{cases} \alpha_8 = del_r(rec \prec b, unify), \alpha_9 = del_r(b \prec rpl, unify) \\ \alpha_{10} = del_a(b, unify) \\ \alpha_{11} = add_a((c, nop, \{x, y, z\}, \{x, y\}), unify) \\ \alpha_{12} = add_r(rec \prec c, unify) \\ \alpha_{13} = add_r(c \prec rpl, unify) \end{cases} \quad \begin{aligned} & (= \alpha_1 = \alpha_5) \\ & (= \alpha_6) \\ & (= \alpha_7) \end{aligned} \\
 S &= (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7, \alpha_8, \alpha_9, \alpha_{10}, \alpha_{11}, \alpha_{12}, \alpha_{13})
 \end{aligned}$$

(c) Sequence S of elementary actions equivalent to α

Figure 6.3: Illustrating the *unify* example: $\alpha = u(\{a, b\}, c)$

Idempotency & Redundant Actions. When looking at the produced elementary action sequences, one can immediately identify redundant actions. For example, in the sequence of actions S produced in FIG. 6.3, several actions are redundant. Even if such redundancy is not an issue according to the intrinsic idempotency of the elementary actions, the sequences can be *optimized*. We represent here the complete sequence, underlining the redundant actions, and then propose an equivalent sequence of actions S' :

$$\begin{aligned}
 S &= (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7, \alpha_8, \alpha_9, \alpha_{10}, \underline{\alpha_{11}}, \underline{\alpha_{12}}, \underline{\alpha_{13}}) \\
 S' &= (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_6, \alpha_7, \alpha_8, \alpha_9, \alpha_{10})
 \end{aligned}$$

We do not address in ADORE the optimization of elementary action sequences, and expose it as a perspective of this work.

Discharge. This composite action allows one to *discharge* the contents of a given *process* into another one (called *target*). As a consequence, the discharged process is destroyed and its contents is added inside the targeted one. We model this action as a term of arity two.

$$d(process, target) \equiv discharge(process, target) \in Action, target \in \mathcal{P}, process \in \mathcal{P}$$

We describe here the elementary actions associated to the *discharge* action. An example of its usage is given in SEC. 6.4.3.

$$\begin{aligned} d(p, p') &\equiv discharge(p, p') \in Action, p \in \mathcal{P}, p' \in \mathcal{P} : \\ del &= \{del_r(r, p) \mid \exists r \in rels(p)\} \cup \{del_i(i, p) \mid \exists i \in polys(p)\} \\ &\quad \cup \{del_a(a, p) \mid \exists a \in acts(p)\} \cup \{del_v(v, p) \mid \exists v \in vars(p)\} \\ add &= \{add_v(v, p') \mid \exists v \in vars(p')\} \cup \{add_a(a, p') \mid \exists a \in acts(p')\} \\ &\quad \cup \{add_r(r, p') \mid \exists r \in rels(p')\} \cup \{add_i(i, p') \mid \exists i \in polys(p')\} \\ \rightsquigarrow &() \stackrel{\vdash}{\vdash} del \stackrel{\vdash}{\vdash} (del_p(p), add_p((p', \emptyset, \emptyset, \emptyset)) \stackrel{\vdash}{\vdash} add \end{aligned}$$

6.4 Defining Algorithms to Interact with Models

The *Separation of Concerns* paradigm allows designers to separate “things” (e.g., concerns). Composition algorithms are then used to *compose* these different “things” into a final system. Several composition algorithms can be found in the literature. Using these algorithms, one can compose class diagrams [Baniassad and Clarke, 2004, Reddy et al., 2006], state-charts [Nejati et al., 2007], sequence diagrams [Klein et al., 2006], ... According to the previously defined action language, we can define composition algorithms dedicated to the ADORE meta-model (e.g., “integrate a fragment inside another process”, CHAP. 7). These algorithms will be formally defined in the next chapters. In this section, we focus on the notation associated to ADORE algorithms, and then illustrate the way algorithms are described on two simple examples: (i) process cloning and (ii) process correlation.

Algorithm Abstract Description. A composition algorithm takes as input everything it is necessary for it to work (e.g., ADORE models, directives, meta-data), and produces as output a sequence of actions. Executing this sequence will produce a new universe where the effect of the algorithm will be visible. Since the sequence of actions is not executed, there is no changes on the original model. According to the logical background of ADORE, we use in the algorithm description a logical-based notation (e.g., quantifiers). We use the \Leftarrow symbol to denote imperative assignment.

6.4.1 Traceability: Algorithm Context & Symbol Generation

Model composition can be compared with model transformations [Baudry et al., 2005]. Model transformation often involves traceability concerns between legacy and newly created artifacts: one must maintain a link between the original entity and the one created while transforming the model [Falleri et al., 2006].

We address this issue in ADORE by providing three functions used to handle links between model elements. While performing a composition, an algorithm can require a unique *context* (modeled as a ground term) by calling the *context()* function. For a given context *ctx* and a set of element e^* , the *gensym*(e^*, ctx) function returns an unique ground term associated to these elements inside *ctx* (reifying a $n \rightarrow 1$ traceability relation). The *gensym* function is idempotent, and associates a single symbol for each element used as input. When the element set contains a single element, we allow the usage of the element $e \in \mathcal{E}$ itself instead of the associated singleton $\{e\} \in \mathcal{E}^*$.

$$\text{Let } e \in \mathcal{E}, ctx = context(), s = gensym(e, ctx) \Rightarrow \nexists e' \in \mathcal{E}^*, e' \neq e, s = gensym(e', ctx)$$

6.4.2 Illustration #1: A Cloning Algorithm

We consider here an algorithm used to compute a *clone* of an ADORE process. We call a *clone* of a process $p \in \mathcal{P}$ a process p' which is *equivalent* to p : $p' \equiv p$. We use this algorithm to illustrate the traceability functions, as it intensively relies on it. The algorithm is described in ALG. 1. It takes as input a process p and the expected name for its clone. It produces as output a sequence of actions which create the clone process, clone all variables, activities, relations and policies.

Algorithm 1 CLONEPROCESS: $p \times p' \mapsto \text{actions}$

Require: $p \in \mathcal{P}$, $p' \in \text{GroundTerm}$

Ensure: $\text{actions} \in \text{Actions}_\perp^*$

```

1:  $ctx \leftarrow \text{context}()$ ,  $\text{actions} \leftarrow (\text{add}_p((p', \emptyset, \emptyset, \emptyset))$ 
2:  $\forall v \in \text{vars}(p)$  : {Variables}
3:    $\text{old} \leftarrow \text{name}(v)$ ,  $\text{new} \leftarrow \text{gensym}(v, ctx)$ ,  $\theta \leftarrow \{\text{old} \leftarrow \text{new}\}$ ,  $\text{actions} \stackrel{+}{\leftarrow} \text{add}_v(v\theta, p')$ 
4:  $\forall a \in \text{acts}(p)$  : {Activities}
5:    $\text{old} \leftarrow \text{name}(a)$ ,  $\text{new} \leftarrow \text{gensym}(a, ctx)$ 
6:    $\theta \leftarrow \{\text{old} \leftarrow \text{new}\} \cup \{v \leftarrow n \mid v \in \text{vars}(a), n = \text{gensym}(v, ctx)\}$ ,  $\text{actions} \stackrel{+}{\leftarrow} \text{add}_a(a\theta, p')$ 
7:  $\forall r \in \text{rels}(p)$  : {Relations}
8:    $\text{old}_l \leftarrow \text{name}(\text{left}(r))$ ,  $\text{old}_r \leftarrow \text{name}(\text{right}(r))$ ,
9:    $\text{new}_l \leftarrow \text{gensym}(\text{left}(r), ctx)$ ,  $\text{new}_r \leftarrow \text{gensym}(\text{right}(r), ctx)$ 
10:   $\theta \leftarrow \{\text{old}_l \leftarrow \text{new}_l, \text{old}_r \leftarrow \text{new}_r\} \cup \{v \leftarrow v' \mid r = a \stackrel{v}{\rightarrow} a' \wedge v' = \text{gensym}(v, ctx)\}$ 
11:   $\text{actions} \stackrel{+}{\leftarrow} \text{add}_r(r\theta, p')$ 
12:  $\forall i \in \text{pols}(p)$  {Iteration Policies}
13:   $\text{old}_* \leftarrow \text{name}(\text{set}(i))$ ,  $\text{new}_* \leftarrow \text{gensym}(\text{set}(i))$ ,  $\text{old} \leftarrow \text{name}(\text{scalar}(i))$ ,  $\text{new} \leftarrow \text{gensym}(\text{scalar}(i))$ 
14:   $\theta \leftarrow \{\text{old}_* \leftarrow \text{new}_*, \text{old} \leftarrow \text{new}\} \cup \{o \leftarrow n \mid \exists a \in \text{acts}(i), o = \text{name}(a) \wedge n = \text{gensym}(a, ctx)\}$ 
15:   $\text{actions} \stackrel{+}{\leftarrow} \text{add}_i(i\theta, p')$ 
16: return  $\text{actions}$ 

```

Since these actions are not executed, the *clone* does not really exist. One can then use the do^+ function to execute this action set and obtain in return an universe containing both processes.

$$\exists p \in \mathcal{P}, \exists u \in \mathcal{U}, do^+(\text{CLONEPROCESS}(p, p'), u) = u' \in \mathcal{U}, \exists \pi \in \text{procs}(u'), \text{name}(\pi) = p' \wedge \pi \equiv p$$

Example. Using as input the simple example depicted in FIG. 6.1, the clone algorithm will produce the sequence of actions S' described in FIG. 6.4. We can notice that this sequence of actions is equivalent to the one used to build the business process (modulo the names used by the elements) depicted in FIG. 6.1(a).

6.4.3 Illustration #2: Naive Process Correlation

We consider here an algorithm used to *correlate*⁶ process execution. This algorithm takes as input two orchestrations $(o_1, o_2) \in \mathcal{O}^2$, and returns the sequence of actions to perform in order to correlate the activities defined in o_2 with the execution of o_1 . This algorithm is a reduced version (*i.e.*, naive) of a composition algorithm used in the context of grid-computing [Nemo et al., 2007]. Contrarily to the original algorithm, we assume that the orchestrations used as inputs only define one *reply* activity.

This naive version of the algorithm is described in ALG. 2. It starts by discharging the content of o_2 inside o_1 . Message reception activities are unified, and response sending activities too. The computed action sequence is then returned to the caller. An illustration of the usage of the NAIVECORRELATION algorithm is depicted in FIG. 6.5

6.4.4 Composition Algorithms Defined using ADORE

In this chapter, we describe the underlying mechanisms used to define algorithms in ADORE. We illustrate this capability by defining two simple algorithms. Since the ultimate goal of ADORE

⁶The processes will share the message reception and response sending, and execute their internal behavior in parallel.

$$\begin{aligned}
example &= (example, \{v\}, \{rec, rpl\}, \{rec \prec rpl\}) \in \mathcal{O} \\
v &= (v, data, false) \in \mathcal{V} \\
rec &= (rec, receive, \emptyset, \{v\}) \in \mathcal{A} \\
rpl &= (rpl, reply, \{v\}, \emptyset) \in \mathcal{A}
\end{aligned}$$

(a) Original Process: $example \in \mathcal{O}$ (FIG. 6.1(b))

$$\begin{aligned}
\text{Let } ctx &= context(), v' = gensym(v, ctx), \\
&\quad rec' = gensym(rec, ctx), rpl' = gensym(rpl, ctx) \\
\alpha_1 &= add_p(example', \emptyset, \emptyset, \emptyset) \\
\alpha_2 &= add_v((v', data, false), example') \\
\alpha_3 &= add_a((rec', receive, \emptyset, \{v'\}), example') \\
\alpha_4 &= add_a((rpl', reply, \{v'\}, \emptyset), example') \\
\alpha_5 &= add_r((rec', rpl', waitFor), example')
\end{aligned}$$

(b) $S' = \text{CLONEPROCESS}(example, example') \in \text{Action}_{<}^*$

Figure 6.4: Illustration of the CLONEPROCESS algorithm

Algorithm 2 NAIVECORRELATION: $o_1 \times o_2 \mapsto \text{actions}$

Require: $o_1 \in \mathcal{O}, o_2 \in \mathcal{O}, |exit(o_1)| = |exit(o_2)| = 1$

Ensure: $\text{result} \in \text{Actions}_{<}^*$

```

1:  $ctx \leftarrow context()$ 
2:  $rec^* \leftarrow \{a \mid \exists a \in (acts(o_1) \cup acts(o_2)), kind(a) = receive\}$ 
3:  $rep^* \leftarrow \{a \mid \exists a \in (acts(o_1) \cup acts(o_2)), kind(a) = reply\}$ 
4: return  $(discharge(o_2, o_1), u(rec^*, gensym(rec^*, ctx), o_1), u(rep^*, gensym(rep^*, ctx), o_1))$ 

```

is to build business process by composing small artifacts into more complex ones, we formally define four algorithms used to handle recurring situations. These algorithms are fully described in the next chapters. We provide here an informal description of each algorithm.

- **WEAVE** (CHAP. 7): this algorithm is used to integrate *fragments* of processes inside others processes. It allows designers to separate *concerns* (e.g., security, business enhancement) into fragments, and then automatically perform the integration of these entities in legacy business processes. This algorithm is *order-independent* and ensures properties such as guard and order preservation.
- **MERGE** (CHAP. 8): this algorithm focus on the *shared join points* issue. The previous algorithm aims to weave a fragment on a given location. When several fragments need to be woven on the same location, *interference* can appear between the different behaviors. The MERGE algorithm composes the different fragments into a single one, ensuring that the result of the composition is order-independent.
- **INLINE** (CHAP. 9): this algorithm is inspired by the compilation domain. It allows to *inline* a process inside another one. As a consequence, at the design level the entities are kept separated, but one can decide to reduce the number of exchanged message in the SOA by using this algorithm.
- **DATAFLOW** (CHAP. 10): this algorithm is inspired by the grid-computing community. It allows to define a process working on a scalar data s , and then automatically enhance it to obtain a process iterating over a set of data $s^* = \{s_1, \dots, s_n\}$.

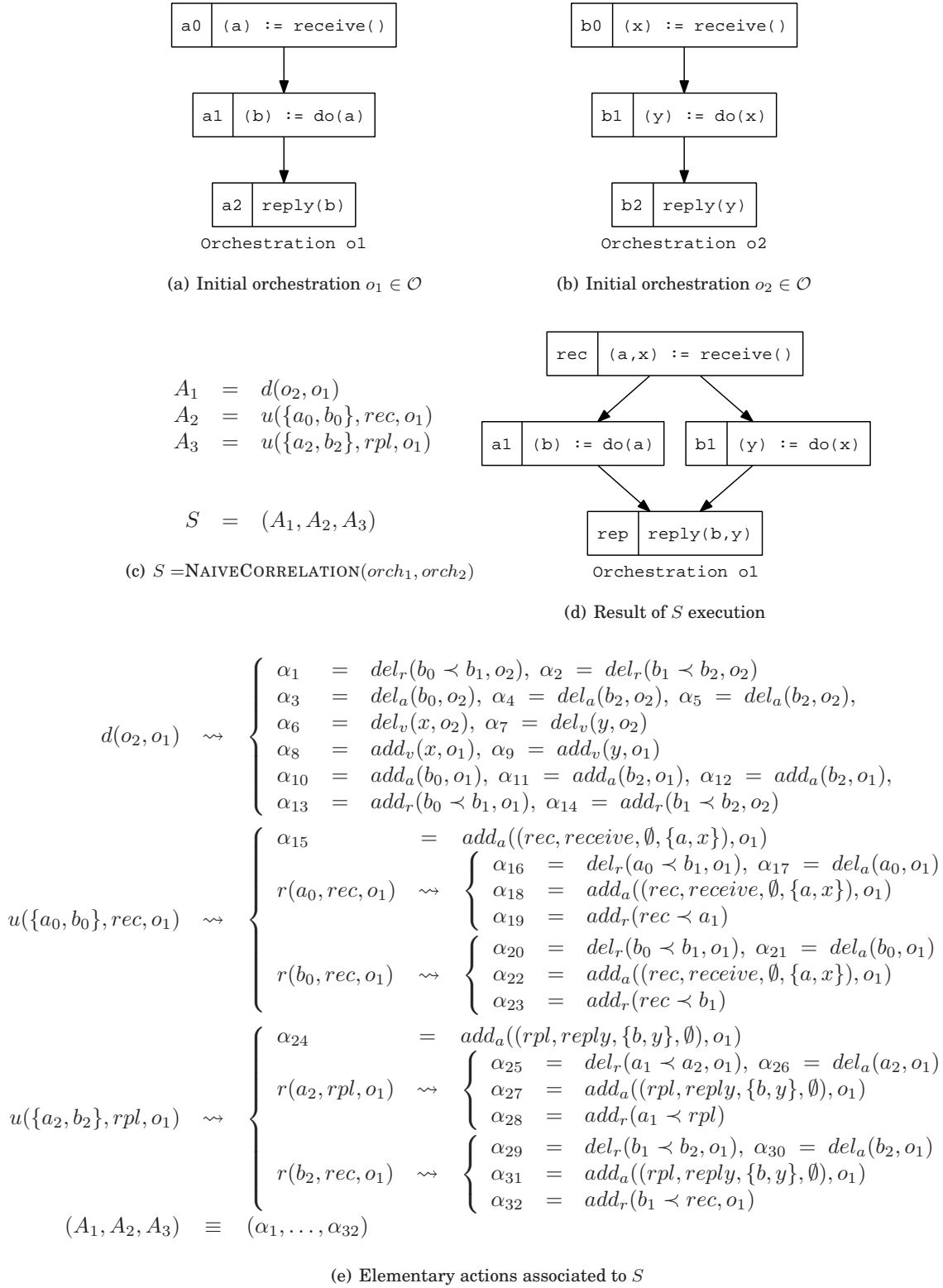


Figure 6.5: Illustrating the NAIVECORRELATION algorithm

In this part, we described in CHAP. 4 a formal meta-model used to support the ADORE approach. The ADORE meta-model allows designers to represent business processes (as orchestration of services) and fragments of business processes. It is defined using many-sorted first order logic as foundations. We made the choice to only present in this document the formalism, and leave the implementation details (the meta-model and the associated tools are implemented in PROLOG & Java and represent 13,000 lines of code) as appendix (CHAP. A).

Thanks to this formalism, we described in CHAP. 5 a graphical syntax used to represent models of business processes. This syntax, inspired by UML activity diagrams, allows one to intuitively understand a process without priori knowledge on ADORE. However, we also described a formal execution semantic associated to the meta-model, anchoring this intuition into a computable set of logical formulas.

Finally, we provided in CHAP. 6 an action framework and the associated execution engine used to interact with ADORE models. Elementary actions (*add* & *del*) and their associated semantics are described, and then composite activities (*replace*, *unify*, *discharge*) are formalized as sequences of elementary actions. This action framework allows the definition of composition algorithms, defined as action sequences producers.

~> Based on these formal foundations and the associated tooling, we can now describe in the next part the four composition algorithms defined to support the ADORE approach.

Part III

Composition Algorithms

Notice: This part defines four composition algorithms defined to support designers while modeling large business processes. The first and second algorithm (CHAP. 7, CHAP. 8) draw a parallel between ADORE and Aspect-Oriented mechanisms (in terms of weaving and merging mechanisms). The third algorithm (CHAP. 9) is inspired by compilation techniques and targets processes performance. Finally, the last one (CHAP. 10) focus on unanticipated iteration introduction (inspired by grid-computing dataflow mechanisms). all along this part, we incrementally define a development approach defined as the composition of these four algorithms.

Integrating Fragments through the WEAVE Algorithm

«Nature uses only the longest threads to weave her patterns, so that each small piece of her fabric reveals the organization of the entire tapestry. »

Richard P. Feynman

Contents

7.1 Introduction	85
7.2 Algorithm Description	86
7.2.1 Principles of the WEAVE Algorithm	86
7.2.2 SIMPLEWEAVE: Integrating a single directive	92
7.2.3 WEAVE: Integrating multiples directives	94
7.3 Behavioral Interference Detection Mechanisms	96
7.3.1 Concurrency Introduction: Variable & Termination	97
7.3.2 Equivalent Activities & Directives	99
7.3.3 Intrinsic Limitation: No Semantic Interference Detection	100
7.4 Properties Associated to the WEAVE Algorithm	100
7.4.1 Order Independence & Determinism	100
7.4.2 Preservation of Paths, Execution Order and Conditions	101
7.4.3 Completeness: Variable Initialization & Process Response	102
7.5 PICWEB Illustration	103
7.5.1 Initial Artifacts: Fragments & Process	103
7.5.2 Weaving a Fragment into Another Fragment	104
7.5.3 Enhancing the <code>getPictures</code> Business Process	104

7.1 Introduction

The ADORE approach allows one to *design* business processes through the usage of the *Separation of Concerns* paradigm. *Fragments* of processes can be automatically introduced into legacy processes. This *weaving* action is realized through a dedicated algorithm, named WEAVE. The goal of this chapter is (i) to describe the mechanisms used to perform a *weave* and (ii) to formally describe the different properties ensured by the algorithm. As ADORE focuses on business process design, we do not address the problematic of dynamic weave (exposed as a perspective of this work). The *weave* algorithm presented here is then *static*, as commonly defined in the literature:

“Static weaving means to modify the source code of a class by inserting aspect-specific statements at join points. For instance, weaving a persistence aspect inserts a database update statement after every assignment to an instance variable.” [Böllert, 1999]

This chapter starts by describing the *weave* algorithm in SEC. 7.2. Considering that a fragment may define a behavior which interferes with the legacy one, we describe in SEC. 7.3 logical rules used to identify such interferences in the models. SEC. 7.4 is dedicated to the description of the properties ensured by the algorithm, such as execution order preservation. Finally, SEC. 7.5 illustrates the WEAVE algorithm on the PICWEB running example.

7.2 Algorithm Description

Using the WEAVE algorithm, a process designer can integrate a fragment $f \in \mathcal{F}$ into another process (*i.e.*, orchestration or fragment) $p \in \mathcal{P}$. To perform the WEAVE, the designer must express *where* the fragment will be woven. This target is defined as a *block* of activities $B \in \mathcal{A}^*$. We also use a binding function β^1 to let the user bind the ghost variables with concrete ones at weave time. We modeled this weaving *directive* as a term ω of arity three:

$$\omega(f, B, \beta) \in \text{Directive}, f \in \mathcal{F}, B \in \mathcal{A}^*, (\beta : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{B}) \in \text{Function}, \exists p \in \mathcal{P}, B \subset \text{acts}(p)$$

To ease the understanding of the WEAVE algorithm, we start by describing a SIMPLEWEAVE algorithm, which only handles a single directive. We define the final WEAVE algorithm as an iteration of the SIMPLEWEAVE one.

7.2.1 Principles of the WEAVE Algorithm

The SIMPLEWEAVE algorithm takes as input a directive $\omega(f, B, \beta)$, and a process p ($B \subset \text{acts}(p)$). The algorithm starts by discharging the content of f in p . As a consequence, all elements (variables, activities and relations) defined in the fragment are now part of the targeted process, including ghost elements (*i.e.*, hook variables and fragment-specific activities). These ghosts must be eliminated from the process. The algorithm relies on two principles to perform this task: (i) it concretises the ghost activities of f (*i.e.*, *predecessors*, *successors* and *hook* are bound with existing elements) and (ii) it replaces the ghost variables of f by real ones according to the binding function β . We start by describing these two mechanisms on simple examples, using a single activity as target. The last paragraph describes how these mechanisms are generalized when targeting a block of activities instead of a single one.

(i) Ghost Activities Vanishment. The ghost activities of the woven fragment must be replaced by the associated elements defined in p , according to B . Considering a directive denoted as $\omega(f, \{a\}, \beta)$, we denote as P_a the set of a predecessors, and S_a the set of a successors in p . The *hook* activity defined in f is denoted as h and its predecessors as P_h (respectively its successors as S_h). When dealing with a successors, we only focus on *control-path* successors². The *predecessor* ghost activity is denoted as \mathbb{P} (respectively \mathbb{S} for the *successors* ghost activity).

$$\begin{aligned} \text{Let } p &\in \mathcal{P}, \omega(f, \{a\}, \beta) \in \text{Directive}, a \in \text{acts}(p), \\ P_a &= \{ \alpha \mid \exists \alpha < a \in \text{rels}(p) \} \in \mathcal{A}^*, S_a = \{ \alpha \mid \exists a < \alpha \in \text{rels}(p) \wedge a \dot{\prec} \alpha \} \in \mathcal{A}^*, \\ \mathbb{P} &= \mathbb{P}(f), \mathbb{S} = \mathbb{S}(f), h = \text{hook}(f) \\ P_h &= \{ \alpha \mid \exists \alpha < h \in \text{rels}(f) \} \in \mathcal{A}^*, S_h = \{ \alpha \mid \exists h < \alpha \in \text{rels}(f) \} \in \mathcal{A}^* \end{aligned}$$

The ghost activities vanishment is accomplished according to the following principles:

- P_1 : **Linking the untargeted behavior with the new one.** Each relation existing in the original process between a predecessor of a ($\pi \in P_a$) and a must be propagated³ to add such relations when a relation is defined between \mathbb{P} and a newly added activity ($\phi \in \text{acts}(f)$). It

¹Usually, the β function is defined as the variable equivalence (\equiv). Using a user-given function allows to let the user choose the variable to use when multiple candidate are equivalents.

²This limitation is described in SEC. 7.2.2. We expose as a perspective a more fine-grained approach to deal with exceptional flow.

³**Propagate**: “the spreading of something into new regions”, <http://wordnet.princeton.edu>.

also propagates the relation existing between a and successors of a ($\sigma \in S_a$), following the same principle.

$$\begin{aligned} & \{add_r((\pi, \alpha, l), p) \mid \exists \pi \in P_a, \exists (\pi, a, l) \in rels(p), \exists \mathbb{P} < \alpha \in rels(f) \wedge \alpha \neq h\} \\ & \cup \{add_r((\alpha, \sigma, l), p) \mid \exists \sigma \in S_a, \exists (a, \sigma, l) \in rels(p), \exists \alpha < \mathbb{S} \in rels(f) \wedge \alpha \neq h\} \end{aligned}$$

\rightsquigarrow The actions generated to perform this task ensure that the legacy predecessors and successors are well-connected with the newly added activities.

- **P_2 : Linking the new behavior with the concrete hook.** Each relation existing between the hook predecessors ($\pi \in P_h$) and the *hook* itself (respectively the *hook* and a successors of the hook — $\sigma \in S_h$) must be propagated to take care of the newly added activities.

$$\{add_r((\alpha, a, l), p) \mid \exists (\alpha, h, l) \in rels(f), \alpha \neq \mathbb{P}\} \cup \{add_r((a, \alpha, l), p) \mid \exists (h, \alpha, l) \in rels(f), \alpha \neq \mathbb{S}\}$$

\rightsquigarrow The actions generated for this task ensure that the activities added through the fragment are well connected with the targeted activity.

- **P_3 : Deleting Ghost Activities.** All relations between \mathbb{P} , \mathbb{S} and h must be deleted. After that these ghost activities can be deleted too.

$$\begin{aligned} & \{del_r((\mathbb{P}, \alpha, l), p) \mid \exists (\mathbb{P}, \alpha, l) \in rels(f)\} \cup \{del_r((\alpha, \mathbb{S}, l), p) \mid \exists (\alpha, \mathbb{S}, l) \in rels(f)\} \\ & \cup \{del_a(\mathbb{P}, p), del_a(h, p), del_a(\mathbb{S}, p)\} \end{aligned}$$

\rightsquigarrow As a consequence, after performing these actions, the ghost elements defined in f are definitively destroyed.

We illustrate this principle in FIG. 7.1, where impacted relations are colored to easily visualize the behavior of the algorithm. In this example, an orchestration o_1 (FIG. 7.1(a)) is enriched by a fragment f (FIG. 7.1(b)), on the activity a_2 (according to the directive depicted in FIG. 7.1(c)). The different “points of interest” (e.g., a_2 predecessors) are described in FIG. 7.1(d). The algorithm starts by discharging f in o_1 (α_1). Then, the actions generated according to P_1 link the legacy behavior to the newly added activities. In the example, activities α_2 to α_5 are generated to link the added activities *bef* and *par* with the legacy predecessors (a_{01} and a_{11}). The new activities are then linked with the concrete target (according to P_2), through the actions α_{10} and α_{11} . Finally (according to P_3), relations involving a ghost activity are removed ($\alpha_{12} \rightarrow \alpha_{17}$), which allow the deletion of these ghost activities (α_{18} , α_{19} & α_{20}).

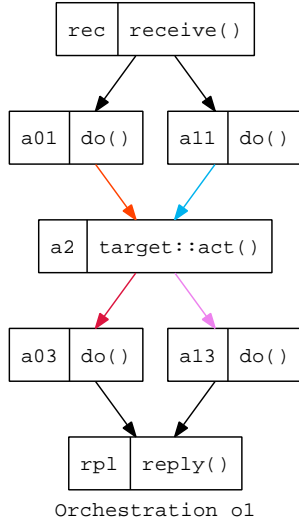
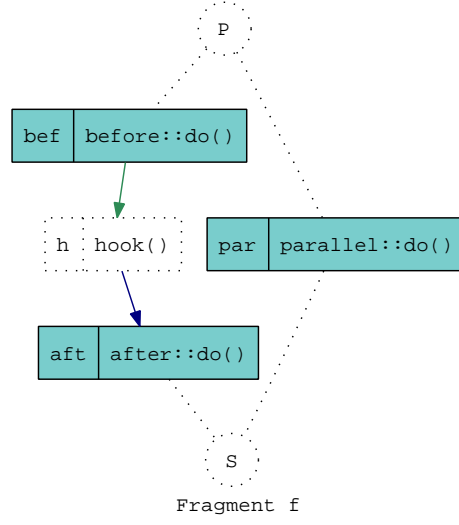
(ii) Ghost Variable Replacement. The ghost variables used in the fragment must be replaced by the concrete ones, preexisting in the legacy behavior. We use the binding function β to perform such a task⁴. For each input ghost variable (respectively output), the β function is invoked to identify a matching candidate existing in the real inputs (respectively outputs) of the targeted activity. Assuming that there is a single candidate matched, the algorithm generates the associated *replace* directives, and picks up the next ghost variable.

$$\begin{aligned} & \{r(v, real, p) \mid \exists v \in inputs(h), \exists ! real \in inputs(a), \beta(v, real)\} \\ & \{r(v, real, p) \mid \exists v \in outputs(h), \exists ! real \in outputs(a), \beta(v, real)\} \end{aligned}$$

The binding process can also identify two other situations: (i) there is no candidate found for this ghost variable or on the contrary (ii) there are multiple candidates. We decide to raise an error and stop the execution of the algorithm when such situations are encountered. Nevertheless, a recent collaboration with the requirement engineering community allows us to consider alternative strategies. This work is exposed as a perspective of this thesis.

We illustrate the ghost variable unification step of the WEAVE algorithm in FIG. 7.2. In this example, the weave directive described in FIG. 7.2(e) asks the algorithm to weave the fragment f' (FIG. 7.2(a)) in the orchestration o_2 (FIG. 7.2(c)), on the activity *tgt*. As the hook activity

⁴We assume that the β function is injective and deterministic

(a) Orchestration $o_1 \in orchs(u)$ (b) Fragment $f \in frags(u)$

Let $S = \text{SIMPLEWEAVE}(\omega(f, \{a_2\}, \equiv), o_1)$, $do^+(S, u) = u'$

(c) Invocation of the algorithm: a_2 is the targeted activity

Let $\omega(f, \{a_2\}) \in Directive$

- $P_a = \{a_{01}, a_{11}\}$
- $S_a = \{a_{03}, a_{13}\}$
- $P_h = \{bef\}$
- $S_h = \{aft\}$

(d) Points of Interest in o_1 and f

$\alpha_1 = d(f, o_1)$

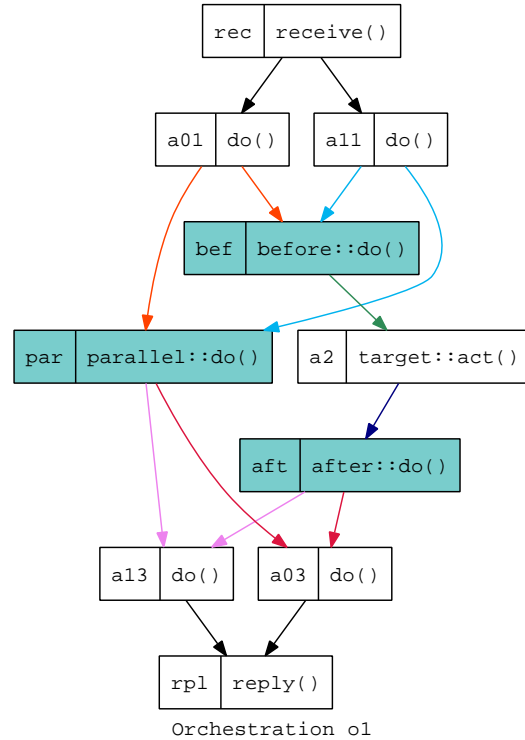
$P_1 : \begin{cases} \alpha_2 = add_r(a_{01} \prec bef, p), \\ \alpha_3 = add_r(a_{11} \prec bef, p), \\ \alpha_4 = add_r(a_{01} \prec par, p), \\ \alpha_5 = add_r(a_{11} \prec par, p), \dots, \end{cases}$

$P_2 : \begin{cases} \alpha_{10} = add_r(bef \prec a_2, p) \\ \alpha_{11} = add_r(a_2 \prec aft, p) \end{cases}$

$P_3 : \begin{cases} \alpha_{12} = del_r(\mathbb{P} < bef), \\ \alpha_{13} = del_r(\mathbb{P} < par), \dots, \\ \alpha_{18} = del_a(\mathbb{P}, p) \\ \alpha_{19} = del_a(h, p) \\ \alpha_{20} = del_a(\mathbb{S}, p) \end{cases}$

$S = (\alpha_1, \dots, \alpha_{20})$

(e) Generated Actions

(f) After the composition: $o_1 \in orchs(u')$ Figure 7.1: WEAVE illustration: Introducing a fragment (f) inside an orchestration (o_1)

defined in f' uses two distinct variables in and out , the algorithm will generate two replacement actions, according to the β function. When dealing with hook input, there is a single variable used as input in tgt and equivalent to it: $in \equiv x$. The associated replacement is then generated: $r(in, x, o_2)$. Considering that the hook output out is equivalent to both z and t variable, and without any extra-information to perform its choice, the algorithm will fail.

$$\begin{array}{ccc} (out, data, false) \equiv (z, data, false) \equiv (t, data, false) & \Rightarrow & \text{"Multiple Candidates!"} \\ out \in outputs(h) & z \in outputs(tgt) & t \in outputs(tgt) \end{array}$$

Letting the user define the β function allows a very powerful expressiveness to perform the binding (relying on the underlying logical model), when the simple variable equivalence \equiv is not sufficient to discriminate the candidates. We will saw in the application chapters (PART IV) that even if this situation is not the main one (*i.e.*, the equivalence is *almost* sufficient), such a mechanism is necessary when dealing with real-life example. In our example, the β function relies on the name of the elements, and supports the generation of the expected replacement directive: $r(out, z, o_2)$.

Handling Blocks of Activities. The previous principles were defined over a *single* concrete activity. Considering the target defined in a directive as a well-formed (PROP. 4.21) activity block $B = \{a_1, \dots, a_n\} \in \mathcal{A}^*$, we can generalize the two previously described principles to handle it as a weave directive target:

- (i) *Ghost activity vanishment.* This step concretises the *predecessors* (denoted as \mathbb{P}), *successors* (denoted as \mathbb{S}) and *hook* (denoted as h) activities. Relations existing between $preds(B)$ and $firsts(B)$ activities (respectively $lasts(B)$ and $nexts(B)$) must be propagated in the newly added behavior according to the relations existing between \mathbb{P} and other activities (respectively others activities and \mathbb{S}). The relations using h as right part (respectively as left part) are propagated for each activity in $firsts(B)$ (respectively $lasts(B)$).
- (ii) *Ghost variable replacement.* We define the inputs of a block as the union of the inputs of its first activities. The outputs are defined as the union of the outputs of its last activities. According to these definitions, the mechanisms described in the previous paragraph stay the same.

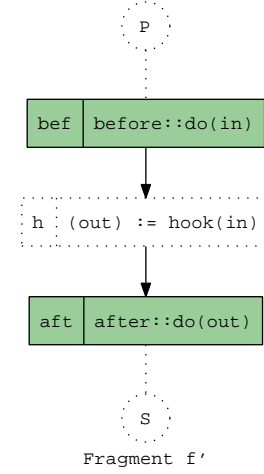
We illustrate the SIMPLEWEAVE algorithm executed using a block as target in FIG. 7.3. In this example, the algorithm weave a fragment f'' (FIG. 7.3(a)) into an orchestration o_3 (FIG. 7.3(c)). According to the weave directive (FIG. 7.3(b)), the target is defined as a block of activities $B \in \mathcal{A}^*$ (colored in gray).

$$\begin{aligned} B &= \{a_{02}, a_{12}, a_3, a_{04}, a_{14}\} \\ firsts(B) &= \{a_{02}, a_{12}\}, lasts(B) = \{a_{04}, a_{14}\} \\ preds(B) &= \{a_{01}, a_{11}\}, nexts(B) = \{a_{05}, a_{15}\} \\ inputs(B) &= \bigcup_{a \in firsts(B)} inputs(a) = \{v_1, v_2\}, inputs(hook(f'')) = \{in\}, v_1 \equiv in \\ outputs(B) &= \bigcup_{a \in lasts(B)} outputs(a) = \{v_3, v_4\}, outputs(hook(f'')) = \{out\}, v_3 \equiv out \end{aligned}$$

We use several colors to identify the way relations are propagated by the algorithm. The composition result is depicted in FIG. 7.3(d).

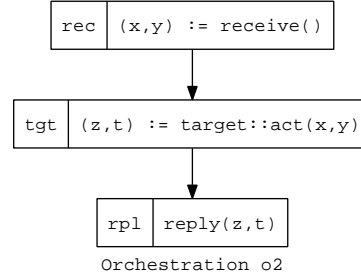
Remark On Activity Synchronization Introduction. Handling blocks of activities as target may enforce the trigger formula associated to an activity. Such a situation happens when several independent branches of activities are used together in the same block, as in FIG. 7.3. Initially, the activity a_{12} waits for the end of the activity a_{11} to start. Since the composition introduces a common ancestor *before* the block, a_{12} waits for the end of this predecessor (*bef*) in the composed process. Since *bef* waits for the end of all the predecessors of the block, it also waits

$$\begin{aligned}
\mathbb{P} &= (p, \mathbb{P}, \emptyset, \emptyset) \in \mathcal{A} \\
bef &= (bef, invoke(before, do), \{in\}, \emptyset) \in \mathcal{A} \\
h &= (h, hook, \{in\}, \{out\}) \\
aft &= (aft, invoke(after, do), \{out\}, \emptyset) \in \mathcal{A} \\
\mathbb{S} &= (s, \mathbb{S}, \emptyset, \emptyset) \in \mathcal{A} \\
in &= (in, data, false) \in \mathcal{V} \\
out &= (out, data, false) \in \mathcal{V} \\
A_{f'} &= \{\mathbb{P}, bef, h, aft, \mathbb{S}\} \in \mathcal{A}^* \\
V_{f'} &= \{in, out\} \in \mathcal{V}^* \\
R_{f'} &= \{\mathbb{P} < bef, bef < h, h < aft, aft < \mathbb{S}\} \in \mathcal{R}^* \\
f' &= (f', A_{f'}, V_{f'}, R_{f'}) \in \mathcal{F}
\end{aligned}$$

(a) Fragment $f' \in \mathcal{F}$ (formal representation)(b) Graphical representation of f'

$$\begin{aligned}
rec &= (rec, receive, \emptyset, \{x, y\}) \in \mathcal{A} \\
tgt &= (tgt, invoke(target, act), \{x, y\}, \{z, t\}) \in \mathcal{A} \\
rpl &= (rpl, reply, \emptyset, \{z, t\}) \in \mathcal{A} \\
x &= (x, data, false) \in \mathcal{V} \\
y &= (y, string, false) \in \mathcal{V} \\
z &= (z, data, false) \in \mathcal{V} \\
t &= (t, data, false) \in \mathcal{V}
\end{aligned}$$

$$\begin{aligned}
A_{o_2} &= \{rec, tgt, rpl\} \in \mathcal{A}^* \\
V_{o_2} &= \{x, y, z, t\} \in \mathcal{V}^* \\
R_{o_2} &= \{rec < tgt, tgt < rpl\} \in \mathcal{R}^* \\
o_2 &= (o_2, A_{o_2}, V_{o_2}, R_{o_2}) \in \mathcal{O}
\end{aligned}$$

(c) Orchestration $o_2 \in \mathcal{O}$ (formal representation)(d) Graphical representation of o_2

$$\begin{aligned}
S &= \text{SIMPLEWEAVE}(\omega(f, \{tgt\}, \beta), o_1) \\
do^+(S, u) &= u'
\end{aligned}$$

$$\begin{aligned}
\beta : \mathcal{V} \times \mathcal{V} &\rightarrow \mathbb{B} \\
v \times v' &\mapsto (v \equiv v') \\
&\quad \vee (\text{name}(v) = \text{out} \wedge \text{name}(v') = z)
\end{aligned}$$

$$r(in, x, o_2) \in S \begin{cases} inputs(h) = \{in\} \\ inputs(tgt) = \{x, y\} \\ \beta(in, x) \quad (in \equiv x) \end{cases}$$

$$r(out, z, o_2) \in S \begin{cases} outputs(h) = \{out\} \\ outputs(tgt) = \{z, t\} \\ \beta(out, z) \quad ("z" = "out") \end{cases}$$

(e) Weave Directive (and consequences)

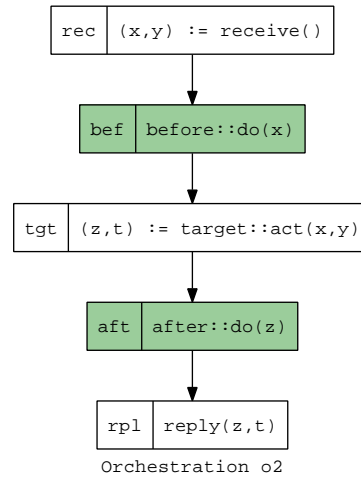
(f) $o_2 \in \text{procs}(u')$

Figure 7.2: WEAVE illustration: Variable replacement

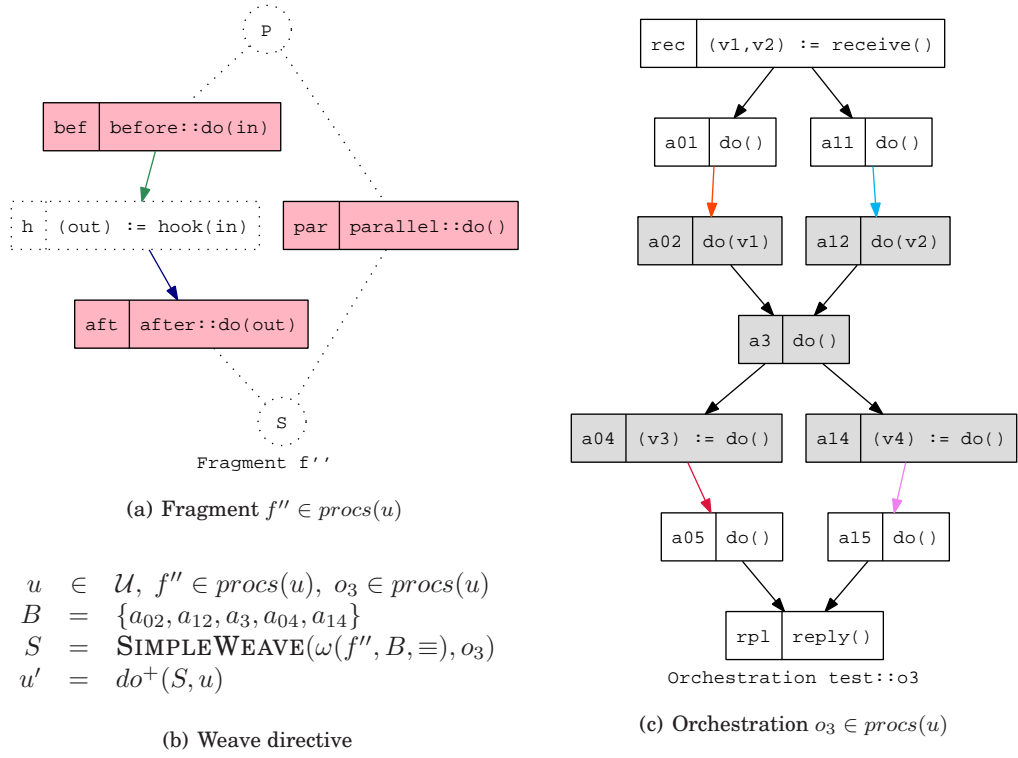


Figure 7.3: WEAVE Illustration: Handling blocks of activities as target

for the end of the a_{01} activity. Consequently, the start of a_{12} is now transitively conditioned by the end of a_{11} **and** the end of a_{01} . In this example, the algorithm introduces two synchronization points (*bef* and *aft* activities). It results in four unanticipated waits: (i) a_{02} waits now for the end of a_{11} , (ii) a_{12} for the end of a_{01} , (iii) a_{05} for the end of a_{14} and (iv) a_{15} for the end of a_{04} . We tackle this issue by assuming that activities used in a block are semantically linked. As a consequence, it makes sense to synchronize the block with all the legacy predecessors (respectively all the legacy successors), even if it was not the case in the preexisting process. The use of process algebras or model checking tools to identify such unwanted synchronizations are exposed as a perspective of this work.

7.2.2 SIMPLEWEAVE: Integrating a single directive

The previous section describes the principles associated to the weave mechanism. We describe in this section the SIMPLEWEAVE algorithm as an implementation of the previously described principles.

Complete Algorithm. We describe in ALG. 3 the SIMPLEWEAVE algorithm. The algorithm starts by initializing local variables and generating the first action: discharging the fragment content into the targeted process (line1). Then, it performs the ghost activity vanishment step (lines 3 \rightarrow 8). The variable replacement step (lines 10,11) is performed before the return of the final action list (line 13). It is important to notice that actions are generated on the legacy processes (*i.e.*, the right part of the intensional definitions relies on the preexisting elements).

Algorithm 3 SIMPLEWEAVE: $\omega \times p \mapsto actions$

Require: $\omega = \omega(f, B, \beta) \in Directive$, $f \in \mathcal{F}$, $B \in \mathcal{A}^*$, $(\beta : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{B}) \in Function$, $p \in \mathcal{P}$

Ensure: $r \in Actions_{\leq}^*$

```

1:                                     { Initialization }
2:  $h \leftarrow hook(f)$ ,  $\mathbb{P} \leftarrow \mathbb{P}(f)$ ,  $\mathbb{S} \leftarrow \mathbb{S}(f)$ ,  $r \leftarrow (d(f, p))$ 
3:  $F_B \leftarrow firsts(B)$ ,  $L_B \leftarrow lasts(B)$ ,  $P_B \leftarrow preds(P)$ ,  $N_B \leftarrow nexts(B)$ 
4:                                     { (i) Ghost Activity Vanishment }
5:  $r \stackrel{\pm}{\leftarrow} \{add_r((\pi, \alpha, l), p) \mid \exists \pi \in P_B, \exists \alpha \in F_B, \exists (\pi, \alpha, l) \in rels(p), \exists \mathbb{P} < \alpha \in rels(f), \alpha \neq h\}$  {P1}
6:  $r \stackrel{\pm}{\leftarrow} \{add_r((\alpha, \nu, l), p) \mid \exists \alpha \in L_B, \exists \nu \in N_B, \exists (\alpha, \nu, l) \in rels(p), \exists \alpha < \mathbb{S} \in rels(f), \alpha \neq h\}$  {P1}
7:  $r \stackrel{\pm}{\leftarrow} \{add_r((a, \alpha, l), p) \mid \exists a \in acts(f), a \neq \mathbb{P}, \exists (a, h, l) \in rels(f), \exists \alpha \in F_B\}$  {P2}
8:  $r \stackrel{\pm}{\leftarrow} \{add_r((\alpha, a, l), p) \mid \exists a \in acts(f), a \neq h, \exists (h, a, l) \in rels(f), \exists \alpha \in L_B\}$  {P2}
9:  $r \stackrel{\pm}{\leftarrow} \{del_r((\mathbb{P}, a, l), p) \mid \exists (\mathbb{P}, a, l) \in rels(f)\} \cup \{del_r((a, \mathbb{S}, l), p) \mid \exists (a, \mathbb{S}, l) \in rels(f)\}$  {P3}
10:  $r \stackrel{\pm}{\leftarrow} (del_a(\mathbb{P}, p), del_a(h, p), del_a(\mathbb{S}, p))$  {P3}
11:                                     { (ii) Ghost Variable Replacement }
12:  $r \stackrel{\pm}{\leftarrow} \{r(v, real, p) \mid \exists v \in inputs(h), \exists ! real \in inputs(B), \beta(v, real)\}$  {Inputs}
13:  $r \stackrel{\pm}{\leftarrow} \{r(v, real, p) \mid \exists v \in outputs(h), \exists ! real \in outputs(B), \beta(v, real)\}$  {Outputs}
14: return  $r$ 
```

Expressiveness Limitations. The algorithm as defined in this section suffers from two expressiveness limitations.

- *Overloading a relation with concrete successors:* According to its definition, the algorithm only propagates relations existing between fragment activities and concrete successors. Consequently, if a new relation (such as a guard) is directly defined in a fragment f between an activity $\alpha \in acts(f)$ and $\mathbb{S}(f)$, the algorithm will not propagate it. This issue is illustrated in FIG. 7.4(a). To tackle this issue, the implementation inserts a *nop* activity between the origin of the relation and $\mathbb{S}(f)$, as illustrated in FIG. 7.4(b).
- *Exceptional Successors:* for a given fragment f , the algorithm assumes that the relation existing between an activity $a \in acts(f)$ and $\mathbb{S}(f)$ is a control-path. Using ADORE, one can intercept an activity in the control-flow, and then enrich its neighborhood, as illustrated in FIG. 7.4(c) and FIG. 7.4(d). But it is not possible to intercept an error throwing. For

example, in FIG. 7.4(c), it is possible to intercept the *thr* activity as any other activity, but it is not possible to intercept “the successors of *tgt* when a fault happens”. We expose this expressiveness limitation as a perspective of this work.

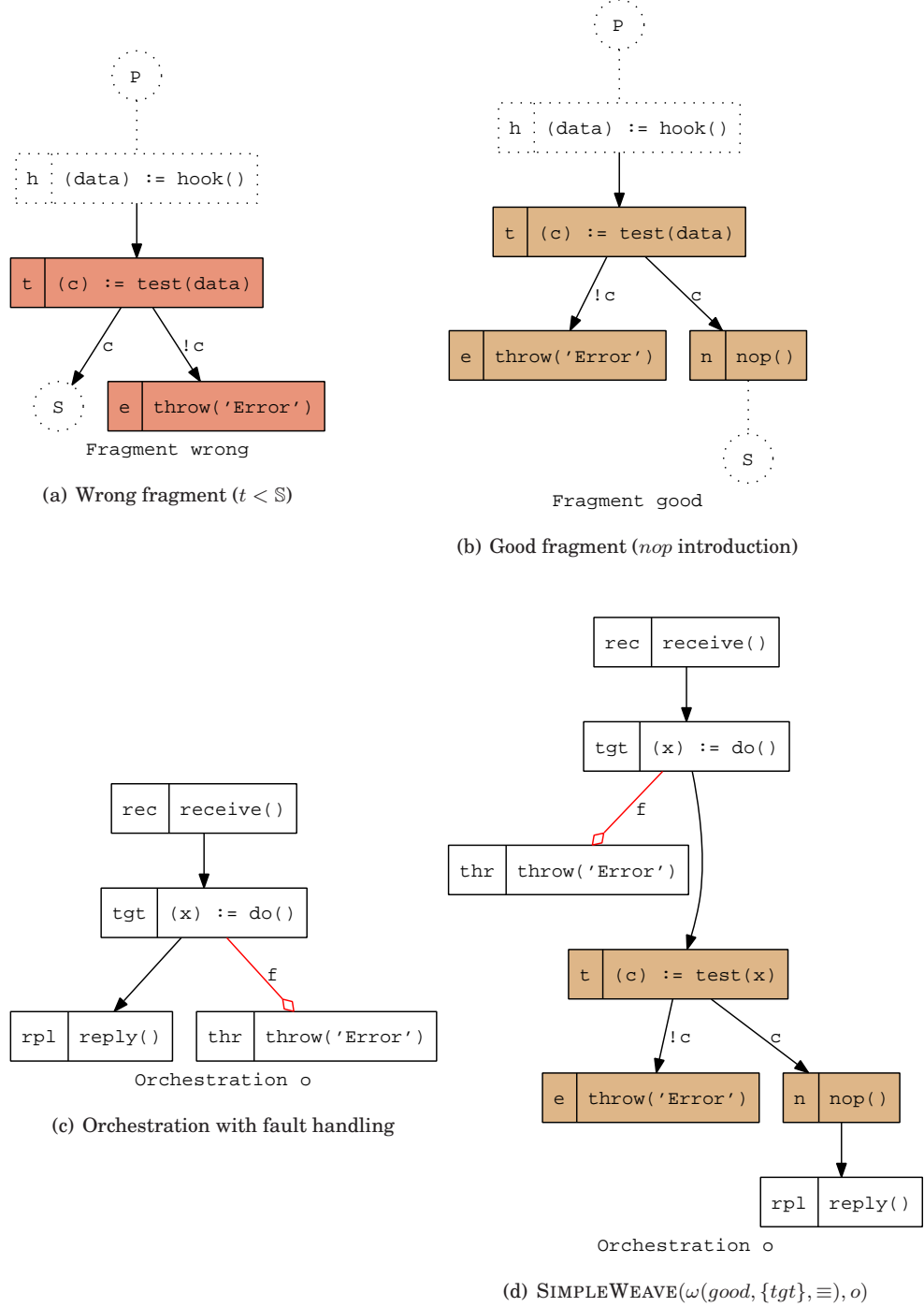


Figure 7.4: Illustrating expressiveness limitations in SIMPLEWEAVE

Remark on Process Simplification. The SIMPLEWEAVE algorithm never deletes preexisting relations. As a consequence, composed processes may contain redundant relations. We made the

choice to *hide* these relations in this chapter, to make the result more readable. Such a situation is depicted in FIG. 7.5, where we depict the *real* process o_2 obtained in FIG. 7.2. We illustrate in FIG. 7.5(b) the two choices made to simplify this process, and the expected “simplified” result is depicted in FIG. 7.5(c). There are two ways to handle this simplification: (i) analyzing the generated action set to identify the redundant actions *a priori* or (ii) defining a cosmetic algorithm and execute it *a posteriori*. We present in APP. B an algorithm able to identify and eliminate redundant relations. An *a priori* detection should be more efficient, which emphasizes the perspective of optimizing the action list, as suggested in the previous chapter.

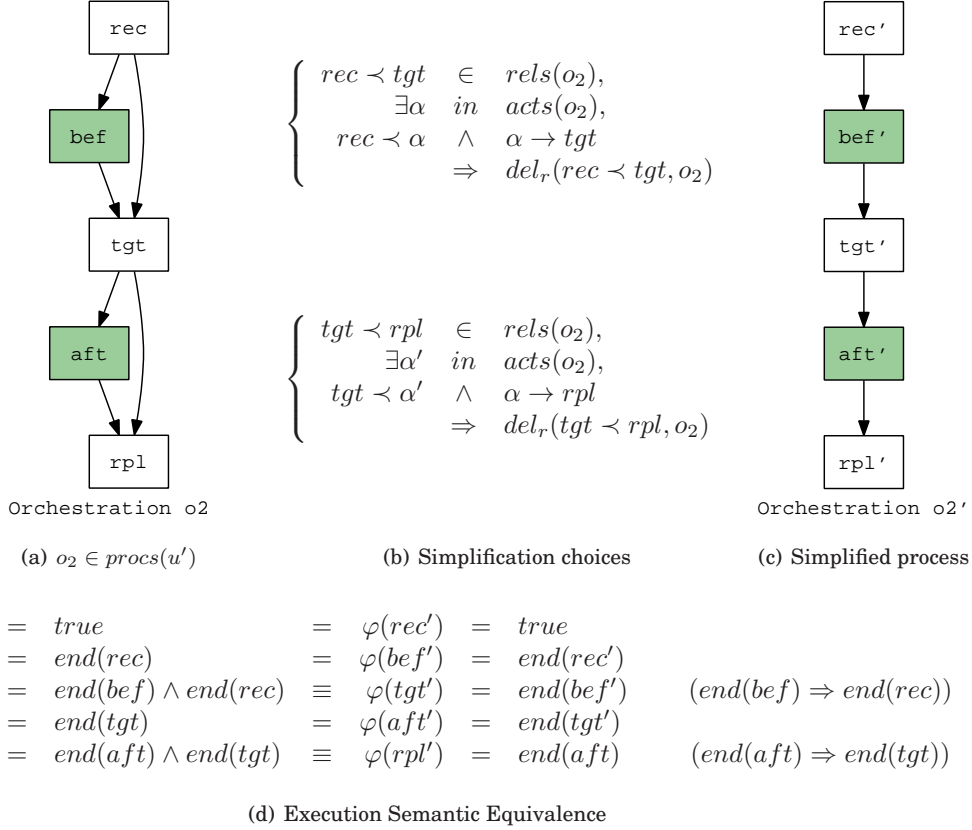


Figure 7.5: Process Simplification: a “cosmetic” need.

7.2.3 WEAVE: Integrating multiples directives

The previously described algorithm only allows the weaving of a single fragment into a single process. We define the final WEAVE algorithm as a folding of the SIMPLEWEAVE one. It takes as input a set of directives⁵ Ω , and the targeted process. The computed action list is returned to the caller as the concatenation of all the action list generated by the SIMPLEWEAVE algorithm.

Algorithm 4 WEAVE: $\Omega \times p \mapsto \text{actions}$

Require: $\Omega \in \text{Directive}^*$, $p \in \mathcal{P}$

Ensure: $\text{result} \in \text{Actions}_{<}^*$

1: **return** $() \stackrel{+}{\vdash} \{\text{SIMPLEWEAVE}(\omega(f, B, \beta), p) \mid \exists \omega(f, B, \beta) \in \Omega\}$

⁵As a consequence, it is not possible to weave the same fragment at the same location more than once. The MERGE algorithm (CHAP. 8) tackles this issue.

Example. We illustrate in FIG. 7.6 the usage of multiple weave directives. In this example, we start by cloning several fragments (named *before* and *after*) before using the clones in a multiple weave⁶. The obtained result is depicted in FIG. 7.6(e). We use colors to easily identify fragment targets and added relations.

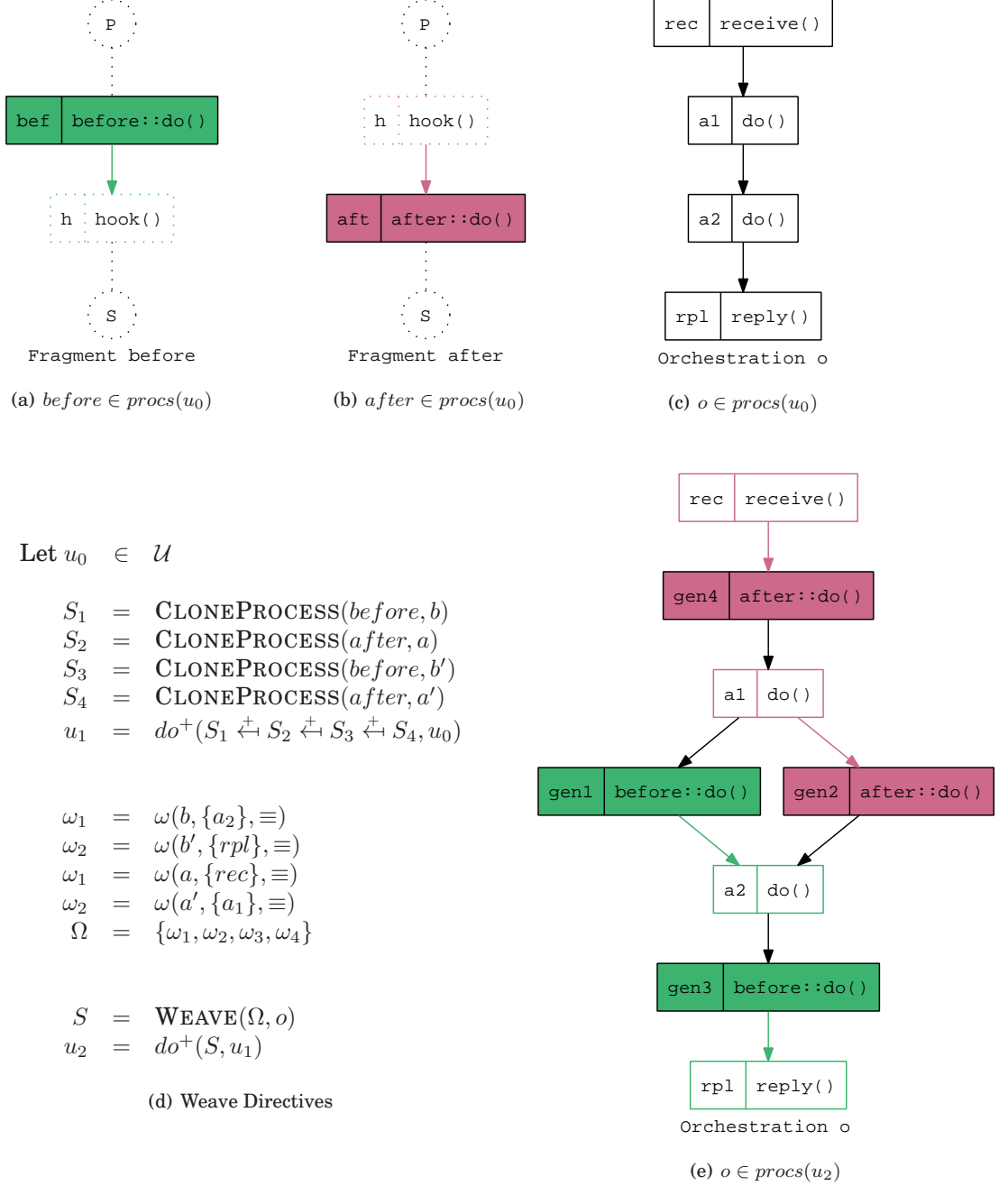


Figure 7.6: WEAVE Algorithm: Illustrating the usage of multiple weave directives

Relations to AOP. An “aspect-oriented” reader will notice that the ADORE weaving mechanisms differ from the one usually defined in AOP frameworks:

⁶According to its definition, a WEAVE will discharge a fragment into a process, and as a consequence destructs it. Using a clone allows one to use the same fragment multiple times.

- *Reduced join points model*: In the AO* vocabulary, join points represent “*well-defined points in the execution of the program*” [Kiczales et al., 2001]. The ASPECTJ framework defines several kinds of join points, at a very fine-grained level: when talking about methods, one can identify and then enhance (i) the method call, (ii) the method call reception or (iii) the method call execution. Considering ADORE as a business process design tool, and business processes as coarse-grained description of workflows, ADORE does not need such a fine-grained expressiveness. Using ADORE, one can simply identify and enhance the execution of a block of activities, according to its intrinsic nature.
- *Pointcut/Advice separation*: Usually, aspect users define *pointcuts* (an expression identifying a set of join points), associated to *advices* (the code to be added at the join point locations), to define *aspects*. Such an approach leads to aspect reusability issues, since an advice is strongly coupled with its associated pointcut⁷. Several approaches propose to decouple an advice from its pointcut [Lagaisse and Joosen, 2006]. ADORE follows the same intention: fragments are defined independently of the weave directive.
- *No pointcut designators*: Usual pointcut designation method introduces hard-coupling between aspects and the base program [Kellens et al., 2006]. It leads to well-known problems such as the *fragile pointcuts* one [Stoerzer and Graf, 2005], where an evolution in the base program may lead to unexpected join points capture. State-of-the-art researches focus on pointcut expressiveness (e.g., LOGICAJ [Rho et al., 2006]). We set up ADORE as a back-end for such approaches. As a consequence, there is no pointcut designators mechanisms in ADORE, and the set of activities targeted for a given weave must be exhaustively described. One can use a pointcut framework to generate such description. We also describe in the CCCMS case study (CHAP. 11) how the ADORE logical back-end can be used to generate weave directives.
- *No aspect ordering*: Several approaches (e.g., ASPECTJ, WEAVER) propose internal mechanisms to order advices between each others [Zhang et al., 2007]. ADORE does not provide such mechanisms, and considers that all fragment are woven *in parallel*. This choice allows us to ensure several properties on the weaving process, such as order-independence. However, aspect ordering can be realized through a chain of weaving (i.e., weave f on o , and then weave f' on the retrieved result). The CCCMS case study (CHAP. 11) intensively uses this mechanism to introduce security concerns in the designed systems: fragments dealing with security are woven on others fragments, which are then used as *secured* versions of the initial concerns they modeled.
- *No before/after keywords*: The ASPECTJ framework makes the distinction between several classes of advices: *around*, *before* and *after*. Using this model, the execution framework starts by executing the first “before” advice, and jumps into the next one until they all start their execution. Then, *around* advices start their execution (according to their user-defined order), followed by the legacy computation (i.e., `proceed` call). The stack of *around* advices is handled, and the execution flow continues. *After* advices are then finally executed, from the last to the first one. ADORE focuses on process *design*, and does not allow such a differentiation between fragments. Fragments can then be seen as *around* advices, without explicit ordering. However, one can simulate the ASPECTJ behavior using ADORE, as shown in APP. C.

7.3 Behavioral Interference Detection Mechanisms

The Separation of Concerns paradigm allows one to define artifacts separately, and then relies on tools to perform the automatic combination leading to the final (composed) system. It is well-known that even if the separate artifacts are correct, interactions between the different concerns may happen and then generate *interference*:

⁷In ASPECTJ, it is possible to avoid this situation through a “tricky” use of aspect inheritance

“Aspects that in isolation behave correctly, may interact when being combined. When interaction changes an aspect’s behaviour or disables an aspect, we call this interference.” [Aksit et al., 2009]

State-of-the-art research work in the AOM research field such as MATA [Whittle et al., 2009] defines interference detection rules, based on critical pair analysis [Heckel et al., 2002]. These mechanisms support designers by analyzing the given directives, and then identify (ii) dependencies (a directive produces an element consumed by another one) and (ii) conflicts (a directive avoid the application of another one). In ADORE, we use similar mechanisms in the *algorithm scheduler* (see SEC. 8.4), at the directive level. In this section describes several mechanisms defined in ADORE to identify interferences at the behavioral level.

7.3.1 Concurrency Introduction: Variable & Termination

Using ADORE, designers define fragments of processes, and then use the WEAVE algorithm to perform the integration of fragments inside legacy process. This software development paradigm supports the design of very large pieces of software, involving several developers working together on the same application. Even if the initial artifacts do not carry concurrent issue, the composition result may encounter two kinds of concurrency issues: (i) concurrent access to a variable and (ii) concurrent response sending.

Activity Exclusiveness. The *concurrent* interference described here intuitively relies on the exclusivity of activities execution. Informally, two activities a and a' are defined as *exclusive* (denoted as $a \otimes a'$) when they cannot be executed in parallel. We base the definition of this property on the boolean formula associated to each activity through ADORE execution semantic (see SEC. 5.3). The property is computed as a comparison of all execution formulas defined in the execution path to identify the exclusion⁸.

$$\begin{aligned} \otimes : \mathcal{A} \times \mathcal{A} &\rightarrow \mathbb{B} \\ (a, a') &\mapsto \begin{cases} \text{Let } p \in \mathcal{P}, e = \text{entry}(p), P_a = \text{path}^+(\text{rels}(p), e, a), P_{a'} = \text{path}^+(\text{rels}(p), e, a') \\ a \neq a', \exists \pi_a \in P_a, \exists \alpha \in \pi_a, \exists \pi_{a'} \in P_{a'}, \exists \alpha' \in \pi_{a'}, \varphi(\alpha) \otimes \varphi(\alpha') \end{cases} \end{aligned}$$

Concurrent Access to a Variable. We consider here an *e-commerce* application (FIG. 7.7), where customers can buy objects on the Web. For illustration purpose, we simplify the orchestration which handles the invoicing process to its smallest expression (FIG. 7.7(a)): it receives a customer c and a price p (*rec*), builds the associated invoice i (*act*) and returns it to the caller (*rpl*). The evolution scenario involves two designers: Bob and Alice. Bob is an *e-commerce* expert and he defines a *discount* fragment (FIG. 7.7(b)) to offer a 10% discount on articles during a special sales period. He asks the fragment to be woven on the *rec* activity. Alice works in the stocks department, and wants to keep a trace of all the prices in a log file. She wrote a fragment named *log* (FIG. 7.7(c)) to perform this task, and asks the fragment to be woven on the invoice building activity (*act*). As a consequence, this scenario involves two different designers who enhance the same process at two different locations, using two different fragments. The obtained process (FIG. 7.7(e)) contains a non-deterministic access to the p variable: there is no order between the activity l and the activity d !

This situation is automatically detected by ADORE through the satisfaction of a logical rule. Such an event is raised to the user, who can identify its source and fix the problem (e.g., by adding a *waitFor* relation between the two activities).

$$\text{Let } p \in \mathcal{P}, \exists a \in \text{acts}(p), \exists v \in \text{outputs}(a), \exists a' \in \text{acts}(p), a' \neq a, v \in \text{vars}(a'), \neg(a \otimes a')$$

Remark: Usual aspect ordering techniques cannot handle this situation, since the two different fragments are woven at different locations in the base process and then cannot be ordered using an ASPECTJ-like mechanisms. Moreover, the fact that ADORE identifies such an issue is

⁸We consider as exclusive two formula φ and φ' (denoted as $\varphi \otimes \varphi'$) since they hold exclusive conditions, or different events for the same activity (i.e., $\text{end}(a)$ and $\text{fail}(a)$).

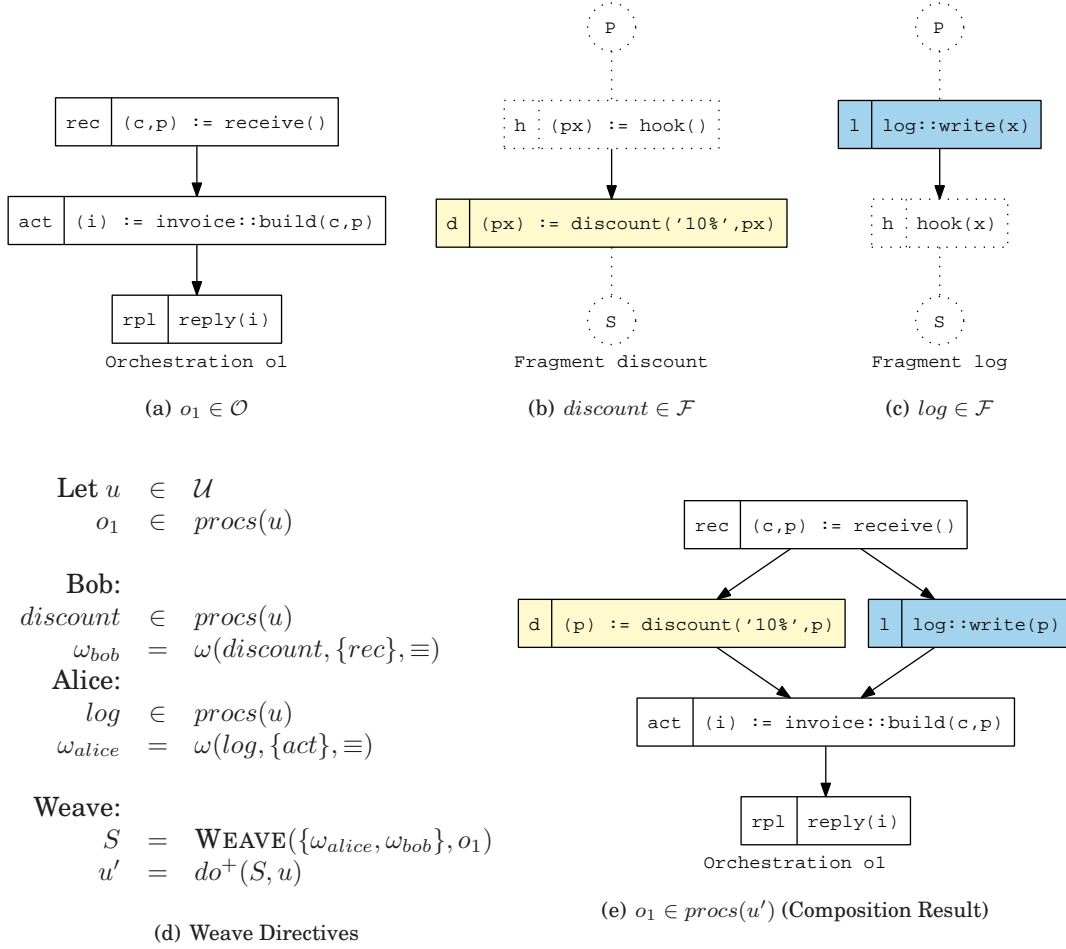


Figure 7.7: Introducing a non-deterministic access to a variable

a strength to support the assessment of large processes built using the separation of concerns paradigm.

Concurrent Termination. The previous paragraph describes how concurrent accesses can be introduced in a process. This paragraph focuses on non-deterministic termination. We use as an example the process depicted in FIG. 7.8(a). It receives two variables a and b (rec), performs a parallel computation (act_1, act_2) to obtain a' and b' , and then replies these values to its caller (rpl). In this scenario, we use two different fragments (possibly defined by two different designers): the first one (FIG. 7.8(b)) is used to check a postcondition on a' (which must be a positive integer), and asks to be woven on act_1 . The second one (FIG. 7.8(c)) ensures a precondition on b (which cannot be equal to zero), and asks to be woven on act_2 . The obtained process is depicted in FIG. 7.8(e). In this process, the two *throw* activities can be executed concurrently if the two conditions are violated, under $(\neg c \wedge c')$ condition set! As a consequence, the composed process is non-deterministic.

This situation is automatically identified by ADORE as the satisfaction of a logical rule. Identifying such a situation supports designers to identify unexpected (and potentially complex) conditions set in their processes.

$$\text{Let } p \in \mathcal{P}, \exists a \in exit(p), \exists a' \in exit(p), a' \neq a, \neg(a \otimes a')$$

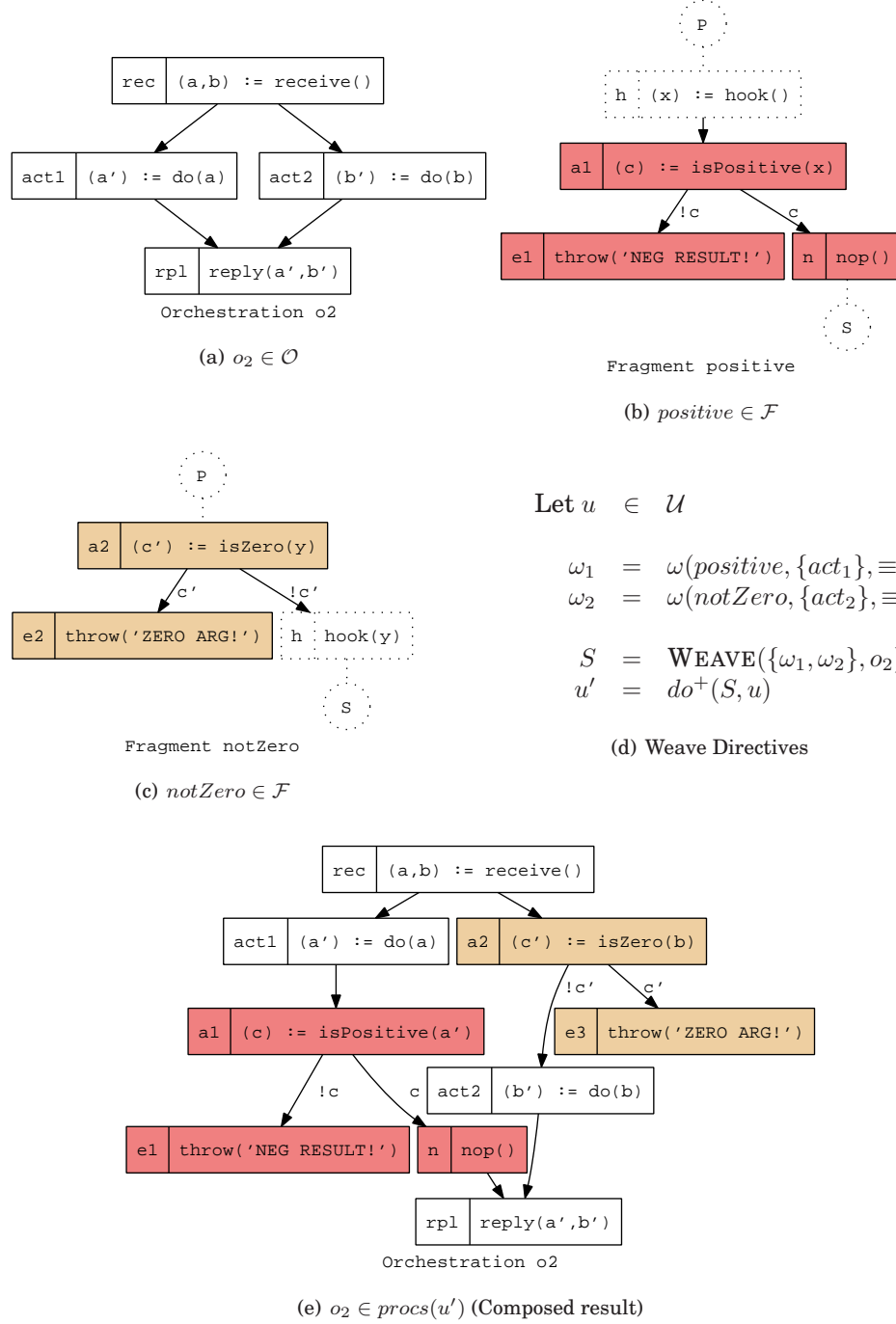


Figure 7.8: Introducing non-deterministic termination in a process

7.3.2 Equivalent Activities & Directives

The previously described interferences were *dangerous* in terms of process execution. We focus now on the detection of *bad-smells* in the composition process. We identified two situations where such bad-smells can occur: (i) a fragment introduces an equivalent activity in a process and (ii) an activity equivalent to one used as a fragment target exists in the process.

Introducing Equivalent Activities. Multiple fragments can introduce service invocations that are equivalent by weaving a same fragment on several activities or by requesting a same service in different fragments. A rule that brings these equivalent services to the attention of the designer can cause the designer to consider how the process can be refactored to avoid unnecessary redundant invocations of services. This situation is illustrated in the CCCMS case study (CHAP. 11).

$$\text{Let } \omega(f, B, \beta) \in \text{Directive}, p \in \mathcal{P}, B \subset \text{acts}(p), \exists \alpha \in \text{acts}(f), \exists \alpha' \in \text{acts}(p), \alpha \equiv \alpha'$$

Forgotten Weave. When a fragment f is woven on an activity a' that is equivalent to another activity a that is not yet woven with f , the designer may have forgotten to weave f on the activity a . Such a situation is identified by ADORE and a suggestion is displayed to the designer. This situation is illustrated in the CCCMS case study (CHAP. 11).

$$\text{Let } \omega(f, B, \beta) \in \text{Directive}, p \in \mathcal{P}, B \subset \text{acts}(p), \exists B' \subset \text{acts}(p) \setminus B, B \equiv B'$$

7.3.3 Intrinsic Limitation: No Semantic Interference Detection

ADORE does not reify the semantic associated to each behavior. As a consequence, the interference detection mechanisms will only detect syntactic interferences. If one tries to weave a fragment which is syntactically coherent with its target but semantically conflicting, such an interference will not be detected by ADORE. We strongly assume that designers know the semantic of the artifacts they are handling. As a consequence, we consider them as responsible of their own acts while building a business process:

“Nemo auditur propriam turpitudinem allegans”⁹.

For example, weaving an encryption fragment and a persistence fragment inside the same process requires to take a particular care of the data when they are not encrypted. Such a preoccupation is semantically driven and it is impossible to detect it syntactically. The use of formal method and knowledge representation such as ontology to automatically analyze and detect interferences in ADORE artifacts at a semantic level are exposed as a perspective of this work.

7.4 Properties Associated to the WEAVE Algorithm

The WEAVE algorithm supports designers when building processes, by automating the integration of fragments. This integration ensures a set of properties, which helps designers to assess the composed process. We classify these properties in three categories: (i) order-independence (to support collaborative design), (ii) elements preservation (to keep preexisting behaviors in the composed system) and (iii) completeness (to ensure that a composed system is still executable). When using the WEAVE algorithm to enrich a preexisting process, designers know that these properties are ensured *by the algorithm*, and can then focus on their business preoccupations.

7.4.1 Order Independence & Determinism

These two properties ensure that the composed process will be the same, independently of the order of the composition directives. This is very important when talking about the collaborative design of business processes, involving several designers.

(P_{w_1}) **Order Independence.** *For a given WEAVE execution, the way designers express the directives does not matter in the obtained result.*

The WEAVE algorithm works on a set of directives Ω , which is by definition unordered. For each directive $\omega(f, B, \beta) \in \Omega$, the algorithm generates a set of actions, but never executes it. Consequently, weave directives are handled *in isolation*, and cannot interfere between each others.

⁹“One cannot invoke in court his own wrongful act.”

(P_{w_2}) **Determinism.** *The WEAVE algorithm is deterministic and ensures to always return the same result for a given set of weave directives.*

The actions generated for each weave directive add new activities and relations in the legacy process, and only delete the ghost elements. The way relations are added is deterministic. According to the idempotency property of the elementary action and the order-independence, the execution of the action set always produces the same process as a result.

7.4.2 Preservation of Paths, Execution Order and Conditions

These two properties ensure that the introduction of a fragment will not “destroy” the execution semantics associated to the existing process.

(P_{w_3}) **Path Preservation.** *An execution of the WEAVE algorithm may extend (e.g., inserting new activities, guards) preexisting execution paths, or add new ones, but cannot destroy existing ones.*

Let f a fragment. According to the *fragment coherence* property (PROP. 4.14), at least one path exists between $\mathbb{P}(f)$, $hook(f)$ and $\mathbb{S}(f)$. Let B the block used as weave target, P_B the preexisting predecessors of B and S_B its preexisting successors. By construction, the $\mathbb{P}(f)$ activity is mapped with all activities defined in P_B , and $\mathbb{S}(f)$ is mapped with all activities defined in S_B . The algorithm never deletes a preexisting relation. Consequently, existing paths are transitively conserved in the composed process.

(P_{w_4}) **Execution Order Preservation.** *If the handled fragment does not bypass their hooks, the preexisting execution order is preserved by construction. In the other situations, such a bypass is considered as expected at the application level since it is defined in the fragment by the designer.*

The path preservation property ensures that the preexisting execution flow is conserved, but does not ensure that the preexisting execution semantic is preserved. The only way of breaking the preexisting semantic is to weave a fragment which defines a path which does not contains its hook activity. Such a situation implies that the concrete successors of the targeted block of activity may then be executed even if the block was not executed, as depicted in FIG. 7.9. In the composed process, the rpl activity can be executed even if the act activity was not executed (contrarily to the base process which ensure that rpl starts only after the end of act). According to ADORE execution semantics, such situation can be added by using *weakWait* relations, fault handling or exclusive conditions to *bypass* the hook. At the implementation level, a warning is raised to the designer when such a situation is detected.

(P_{w_5}) **Condition Preservation.** *Using ADORE, guards added by fragments and preexisting guards cannot interfere between each others and are preserved by construction, since the designer does not use a ghost variable in a guard.*

The previous property ensures that the execution order is preserved. Even if we assume the respect of this property in this paragraph, it does not mean that preexisting conditions are always preserved. According to the process isolation property (PROP. 4.15), each variable defined in a process is unique, and contained by only one process. Variables interact between each others through the *ghost variable replacement* step of the WEAVE algorithm, where *hooked* variables are mapped to concrete ones. As a consequence, the only way for a fragment to interact with a preexisting guard is to define a new guard based on a *hooked* variable, as depicted in FIG. 7.10. In the legacy process, the rpl_1 activity will only start if the c condition is evaluated to true. But in the composed process, since the woven fragment adds the opposite condition to its successors, it implied that the rpl_1 activity will be executed even if c was evaluated to false. A restriction of ADORE expressiveness (a guard relation should not use as left part a ghost variable) ensures that the preexisting conditions are preserved, since the execution order is preserved.

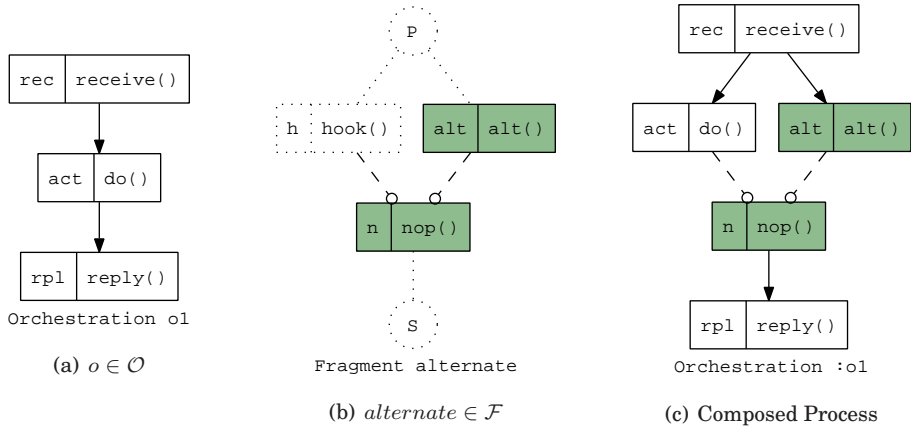


Figure 7.9: Breaking the existing execution semantics through a fragment

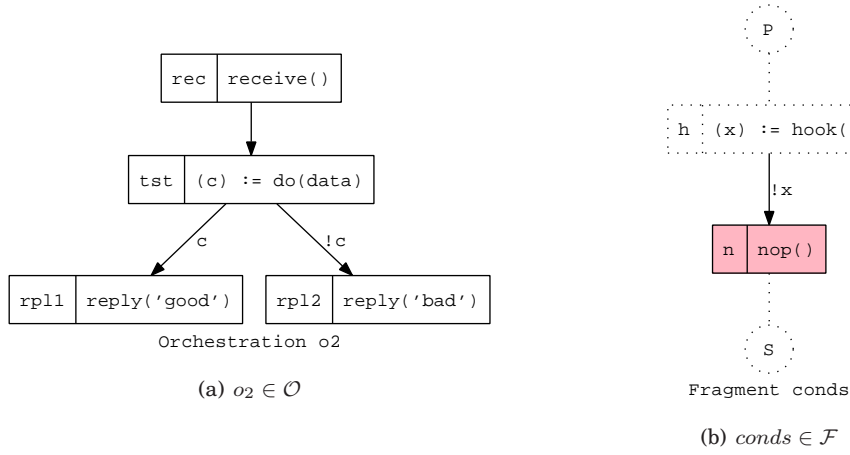


Figure 7.10: Breaking the existing guards relation semantic through a fragment

7.4.3 Completeness: Variable Initialization & Process Response

These properties ensure that the composed process can be executed without any syntactic issue.

(P_{w_6}) **Variable Initialization.** *Assuming that preexisting elements are initialized, the variables handled in a composed process are initialized too, by construction.*

For a given fragment f , ghost variables are the only one which do not need to be initialized: they are replaced with concrete elements by the algorithm. Assuming that the handled processes (both fragment and targeted process) always initialize a variable before using it, the algorithm ensures that no uninitialized variable can be encountered in a composed process (thanks to the path preservation property P_{w_3}). At the implementation level, the verification of variable initializations relies on an exploration of all possible execution paths, which is a very time-consuming operation. As a consequence, it is faster to check this property on small artifacts, and ensure that the composed (eventually huge) process is still valid.

(P_{w_7}) **Process Response.** *For each received message, a response message will be generated by the composed process*

The *entry-to-exit* property (PROP. 4.12) ensures that a fragment f always defines a path between its entry point $\mathbb{P}(f)$ and an exit point $\mathbb{S}(f)$, a *throw* or a *reply* activity). This property must also be true in the targeted process. The algorithm binds $\mathbb{P}(f)$ and $\mathbb{S}(f)$ to concrete activity. As a consequence, the newly added activities are connected to the process entry point (through $\mathbb{P}(f)$ binding) and to the preexisting exit point through $\mathbb{S}(f)$ binding (if needed). Consequently, we can ensure that each activity (which is not an exit point) is connected to at least one exit point in the composed process.

7.5 PICWEB Illustration

In this section, we describe the usage of the WEAVE algorithm to design the PICWEB business process. It illustrates the algorithm capabilities in front of a *real-life* example.

7.5.1 Initial Artifacts: Fragments & Process

We consider as base process the realization of the *getPictures* behavior depicted in FIG. 5.7(a) (p. 55). This orchestration receives a *tag* and a *threshold*. It asks for an API *key* associated to a FLICKR® account through a *registry* service. Then, the FLICKR® service is invoked with the received *tag* and the freshly retrieved *key*. The obtained set of images *picture** is then truncated according to the user-given threshold, and finally returned to the caller.

Based on this implementation of this business process, we focus on the realization of the following concerns: (i) including PICASA™ as a newly available source of pictures, and (ii) take care of a possible timeout from a picture source.

Including PICASA™. This concern is realized by the fragment depicted in FIG. 7.11. This fragment *hooks* an activity (h) which consumes a *flag* and produces a set of data *res**. In parallel of this hook, the fragment calls the PICASA™ service (aP_1) using the hooked *flag*, to retrieve the set of picture available in the PICASA™ repository (in the *pic** variable). Finally, the two datasets are merged (aP_2), and the preexisting execution flow can continue.

Handling a timeout issue. This concern is realized by the fragment depicted in FIG. 7.12. In parallel of the hooked activity (h , producing a *data** variable), it starts a stopwatch, for 60 seconds (t_1). We use *weakWaits* relation to define an alternative path: if the stopwatch reaches zero before the end of the hooked activity, the expected *data** variable is initialized with an empty list (t_2).

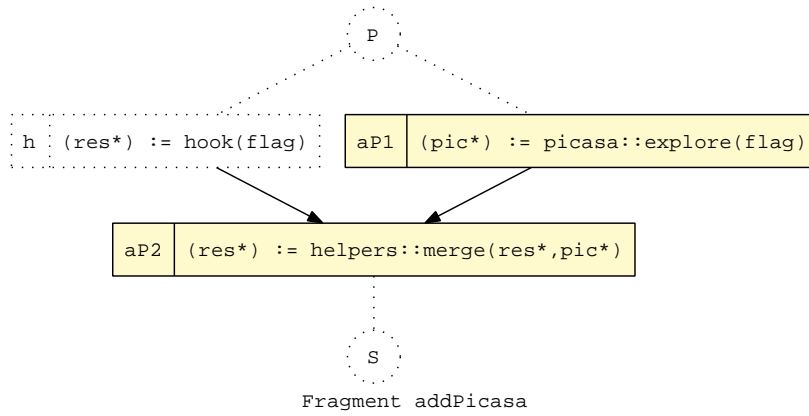
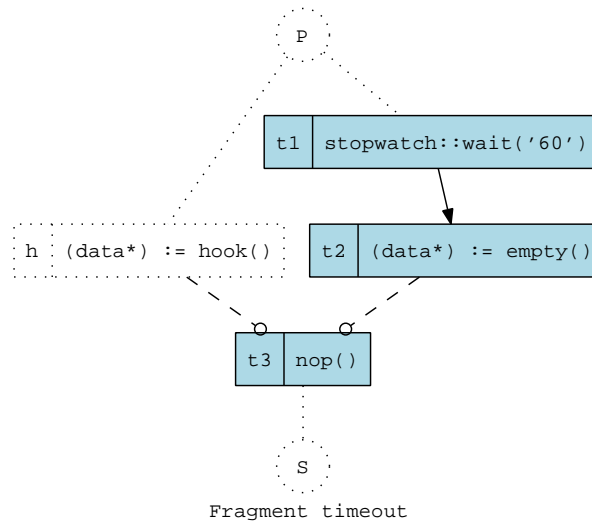


Figure 7.11: Graphical representation of the addPicasa fragment

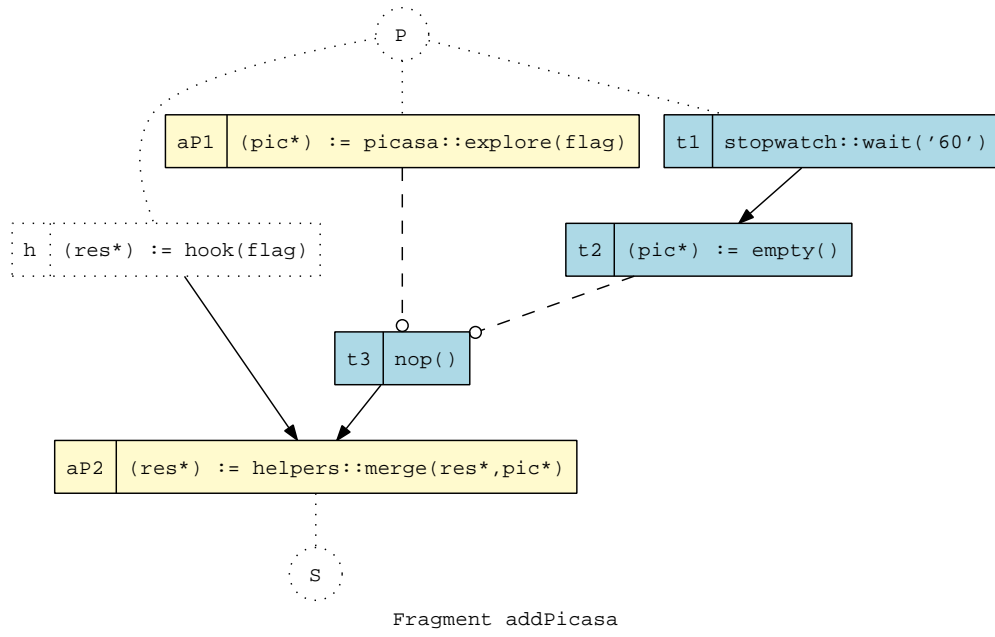
Figure 7.12: Graphical representation of the *timeout* fragment

7.5.2 Weaving a Fragment into Another Fragment

We consider here the following requirement: “an eventual timeout of the PICASA™ source implies to silently ignore the situation”. Since this requirement is defined *globally*, we decide to weave the *timeout* fragment into the *addPicasa* one. Consequently, each usage of the *addPicasa* fragment will include the timeout concerns. The obtained result is depicted in FIG. 7.13.

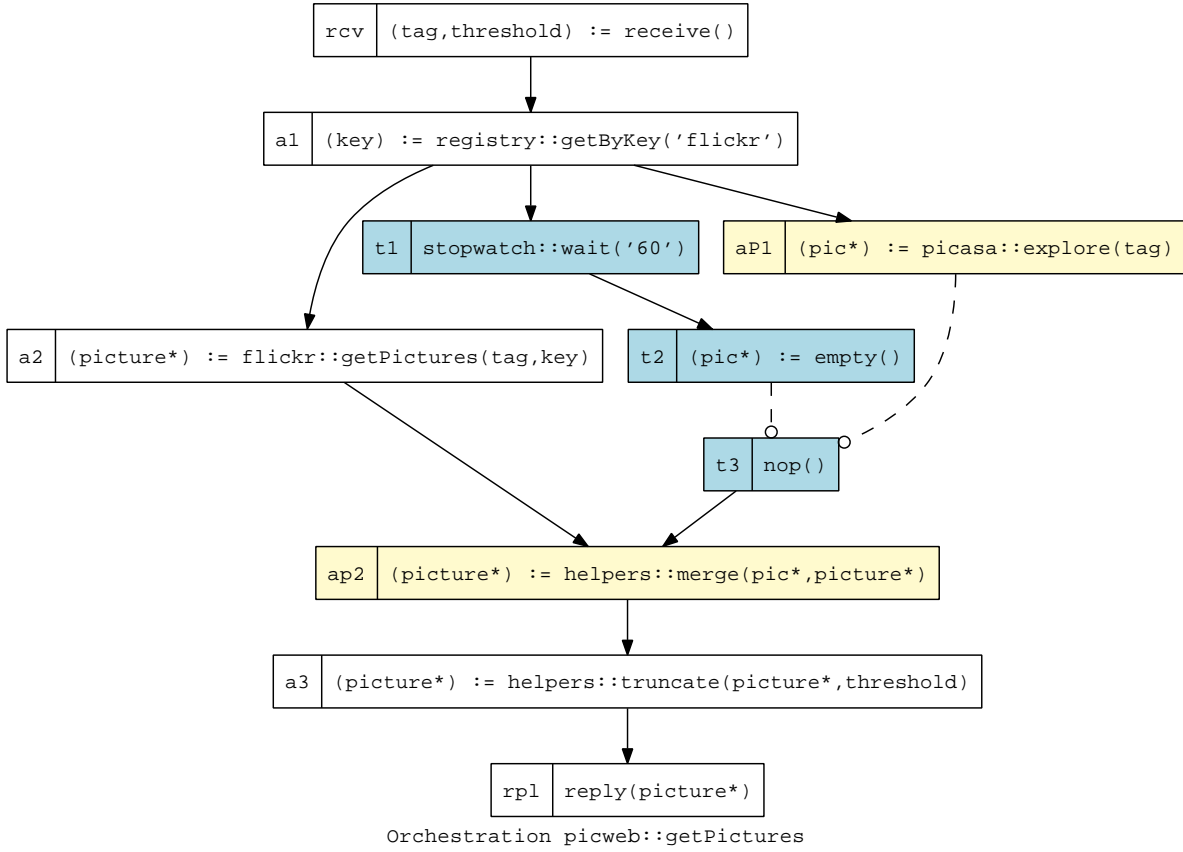
7.5.3 Enhancing the getPictures Business Process

Using the composed fragment built in the previous section, a designer can now integrate the PICASA™ source into the legacy *getPictures* process. As a result (FIG. 7.14), the composed process realizes the legacy behavior, and also call the PICASA™ service, including a mechanism to handle an eventual timeout.



Let $u \in \mathcal{U}$, $addPicasa \in procs(u)$, $timeout \in procs(u)$
 $S = \text{WEAVE}(\{\omega(timeout, \{aP_2\}, \equiv)\}, addPicasa)$, $u' = do^+(S, u)$

Figure 7.13: Enhanced *addPicasa*, including the *timeout* concern



Let $u_0 \in \mathcal{U}$, $addPicasa \in procs(u_0)$, $timeout \in procs(u_0)$

$S_0 = \mathbf{WEAVE}(\{\omega(timeout, \{aP_2\}, \equiv)\}, addPicasa)$, $u_1 = do^+(S_0, u_0)$

Let $getPictures \in procs(u_1)$, $addPicasa \in procs(u_1)$

$S_1 = \mathbf{WEAVE}(\{\omega(addPicasa, \{a_2\}, \equiv)\}, getPictures)$, $u_2 = do^+(S_1, u_1)$

Figure 7.14: Composed `getPictures`, including `addPicasa` and `timeout`

MERGE concerns around *Shared Join Points*

«Do the difficult things while they are easy and do the great things while they are small. A journey of a thousand miles must begin with a single step.»

Lao Tzu

Contents

8.1 Introduction	107
8.2 Algorithm Motivations: “Why not simply ordering?”	108
8.2.1 Step–Wise: Order as the Keystone of Feature Compositions	108
8.2.2 Aspect–Ordering Should Not Be a Technical Answer	108
8.2.3 Ensuring a Deterministic Composition Mechanism	109
8.2.4 Identifying Shared Join Points in Collaborative Design	110
8.3 The MERGE Algorithm	110
8.3.1 Principles	110
8.3.2 Illustrating Example	111
8.3.3 Algorithm Description & Properties	112
8.4 Taming Process Design Complexity	113
8.4.1 Prerequisite: An Algorithm Invocation Pool (\mathbb{I})	113
8.4.2 Principle: Automated Reasoning on the Invocation Pool	113
8.4.3 Storing Designers Choices in a Knowledge Basis (Ξ)	114
8.5 Modeling PICWEB ...	114
8.5.1 The Initial PICWEB System: FLICKR®, PICASA™ & Timeout	114
8.5.2 Introducing Fault Management	115
8.5.3 Step–Wise Development: Adding Cache Mechanisms	117

8.1 Introduction

In the previous chapter, we described the WEAVE algorithm, which supports the integration of fragments inside preexisting processes. It consumes a set of weave directives as input, and produces the sequence of actions needed to perform the expected integrations. It naturally allows several directives to share the same activity block as target, and treats these directives as independent. However, considering a collaborative development context, (where several designers work simultaneously on the same system), this situation should be detected and designers informed: they are trying to enhance a process with different concerns, at the same location. The AO* community identifies this issue under the name of *Shared Joint Point* (SJP).

“It is possible that not just a single, but several units of aspectual behavior need to be superimposed on the same join point. Aspects that specify the superimposition of these units are said to “share” the same join point.”

[Nagy et al., 2005]

In this chapter, we focus on the following problem: “How can one integrate several fragments at the same location in the base program?”. Where usual Separation of Concerns techniques provides *order-driven* techniques, we choose an opposite approach, based on a MERGE algorithm. We discuss in SEC. 8.2 the motivations of this choice. The algorithm is then described in SEC. 8.3. Based on this algorithm and the WEAVE one, we implement in SEC. 8.4 a development methodology which supports designers while performing simultaneous compositions. This method is finally applied to the PICWEB running example in SEC. 8.5

8.2 Algorithm Motivations: “Why not simply ordering?”

8.2.1 Step-Wise: Order as the Keystone of Feature Compositions

Step-wise development relies on ordering mechanisms, by essence. In such an approach, features are composed sequentially, *step by step*. The AHEAD framework [Batory et al., 2004] uses mathematical function composition mechanism ($f \bullet g(x) = f(g(x))$) to compose features into a global system. This approach is natural and intuitive: if one wants to add a *security* feature into a base program p , he/she just needs to express the associated composition: $p' = \text{security} \bullet p$. Each composition defines a *step* in the development process.

This approach makes designers think about the *features* they design, as well as the way these features needs to be ordered. Considering a feature dealing with *persistence* and another one dealing with *security*, one can decide to secure the persistence feature using the following composition: $\text{security} \bullet \text{persistence}$. Even if syntactically correct, the opposite composition $\text{persistence} \bullet \text{security}$ may lead to semantic-driven issue (e.g., “persistent password”). Mathematics techniques such as commuting diagrams can be used to identify and assess feature composition order [Batory, 2008]. For example, in [Kim et al., 2008], authors explore the ordering problem to identify type-safe composition. When the different features are independent, the composition order does not matter. Let f_1 and f_2 two features, and p a program. If we consider f_1 and f_2 as independent (i.e., these two features do not interact with the same element), the composition order is not important: $\text{independent?}(f_1, f_2) \Rightarrow f_1 \bullet f_2 \bullet p = f_2 \bullet f_1 \bullet p$

↪ ADORE defends a slightly different position, which is complementary to the ones described in the *step-wise* paradigm. We consider a *step* with a more coarse-grained definition: a set of concerns of *equivalent* priority needs to be integrated into the legacy architecture. This approach is common according to agile development methodology. In these methodologies¹, the step notion is reified as a *sprint*, i.e., “a time-boxed period of software development focused on a given list of goals”. It lengths approximatively a month, used by the *team* to fulfill the goals they had accepted. ADORE supports designers **inside** a step (FIG. 8.1), when the composition of concerns with the same *priority* (i.e., integrated in the same step) generates unanticipated interferences.

8.2.2 Aspect-Ordering Should Not Be a Technical Answer

Major AO* techniques rely on *aspect-ordering* to handle shared join points. Using these techniques, one can declare precedence rules between several aspects, and then explicitly define the way the different aspects will be executed at runtime. Implementation of AO* approaches such as ASPECTJ [Kiczales et al., 2001] rely on this mechanism. In this framework, the order can be defined (i) explicitly with the **declare precedence** construction or (ii) implicitly, depending the way the different aspects are written. Using aspect ordering to solve an interaction spawn by a

¹For concision reasons, we use here the vocabulary defined by the SCRUM method [Schwaber and Beedle, 2001].

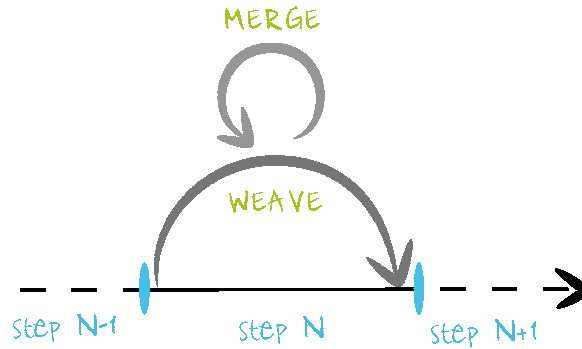


Figure 8.1: Development approach: The MERGE algorithm, supporting the WEAVE one.

shared join point is a coarse-grained resolution method. Let α and α' two aspects, and a precedence relation $precedes(\alpha, \alpha')$. This precedence declaration implies that the advice associated to α will be located before the advice code associated to α' in the aspect execution chain (see APP. C). Such a mechanism is defined as *mandatory* by state-of-art research about AO* development:

“Ordering Aspects – To ensure the required behavior of the superimposed aspects at SJPs, it **must** be possible to specify the execution order of the aspects.” [Nagy et al., 2005]

With this mechanism, two approaches can be used to define a system:

- A_1 . One can work on a given system, identify relations between aspects and then decide of an order, following the same idea than the *step-wise* approach (but there is no composition order exploration support).
- A_2 . Another approach is to perform the weaving, identify bugs in the *aspectized* system, and use aspect-ordering to patch it.

There is no intrinsic difference between these two opposite approaches in state-of-the art implementation of AO* mechanisms.

↪ We defend that using aspect-ordering mechanisms to patch aspect interferences is a technical answer to a badly-stated problem. If two aspects needs to be explicitly ordered (A_1), they should not belong to the same development step. But if several aspects interfere in a given step (A_2), this information **must** be raised to the designers, who can then focus on the understanding on the interference, at the aspect level (and not the composed system one). In ADORE, SJP are automatically identified. We define the MERGE algorithm to support designers in front of SJP. The shared fragments are *merged*, and then returned to the designer who can work on the merge fragment to solve interferences, if any. One can found in the literature others alternative mechanisms to handle shared join point (e.g., [Douence et al., 2004, Lopez-Herrejon et al., 2006]).

8.2.3 Ensuring a Deterministic Composition Mechanism

The ASPECTJ framework makes the choice to accept non-deterministic behavior when no order are defined:

“Otherwise, if two pieces of advice are defined in two different aspects, it is undefined which one has precedence.”

<http://www.eclipse.org/aspectj/doc/next/progguide/semantics-advice.html>

At runtime, the framework will implicitly choose an execution order between the two aspects, due to its sequential handling of aspects. This decision is hidden to the developer, and may lead to the introduction of unanticipated interactions (see APP. E).

- ↪ We defend that the composition process must ensure a deterministic result while handling several fragments at the same join point. The reification of a MERGE algorithm and its associated semantic in ADORE tackles this issue.

8.2.4 Identifying Shared Join Points in Collaborative Design

ASPECTJ does not inform its users when a shared join point is encountered. The underlying dynamic model can explain this decision: aspects can be triggered at runtime according to a dynamic pointcut declaration. ADORE restricting itself to static weaving, a static identification of the shared join points is possible. This information can then be raised to the designers, requiring a particular attention when their fragments must be woven at the same location than others.

- ↪ In ADORE, shared join points are automatically detected, and designers informed. A *merged* fragment can be obtained as a default behavior, through the usage of the MERGE algorithm. Designers can decide to refuse this *default* fragment, and interact together to design an optimized one.

8.3 The MERGE Algorithm

This section describes the principles of the MERGE algorithm, and then provides a description of the algorithm.

8.3.1 Principles

Identifying Shared Join Points. The previously defined WEAVE algorithm (CHAP. 7) uses as input a set of binding directives $\Omega \in \text{Directive}^*$. The algorithm does not reason about the given directive set, and generates the actions associated to each directive. When several designers work together on the same system, the most simple way to integrate their decisions in the final system is to perform the union of their respective directives (SEC. 7.3). Through the analysis of the Ω directive set, it is possible to automatically identify shared join points.

We define the two following functions to identify shared join points in a given set of directives. The *isShared?* boolean predicate allows one to check if a fragment target is shared with other ones. The *shared* function returns the set of fragments woven at the same location B for a given set of directives Ω .

$$\begin{aligned}
 \text{isShared?} : \mathcal{A}^* \times \text{Directive}^* &\rightarrow \mathbb{B} \\
 (B, \Omega) &\mapsto \exists \omega(f, B, \beta) \in \Omega, \exists \omega(f', B, \beta') \in \Omega, f \neq f' \\
 \text{shared} : \mathcal{A}^* \times \text{Directive}^* &\rightarrow \mathcal{F}^* \\
 (B, \Omega) &\mapsto \{f \mid \text{isShared?}(B, \Omega) \wedge \exists \omega(f, B, \beta) \in \Omega\}
 \end{aligned}$$

Fragment Activities Unification. We denote as $F = \{f_1, \dots, f_n\} \in \mathcal{F}^*$ a set of fragments shared around the same join point. The *merged* fragment is defined as the union of the fragments (discharged in a new fragment f'), where *predecessors*, *successors* and *hooks* are unified. We denote as *ctx* the *context* used to generate new symbols.

$$\begin{aligned}
 \text{Let } P &= \{p \mid \exists f \in F, p = \mathbb{P}(f)\}, S = \{s \mid \exists f \in F, s = \mathbb{S}(f)\}, H = \{h \mid \exists f \in F, p = \text{hook}(f)\} \\
 &\rightsquigarrow (u(P, \text{gensym}(P, \text{ctx}), f'), u(S, \text{gensym}(S, \text{ctx}), f'), u(H, \text{gensym}(H, \text{ctx}), f'))
 \end{aligned}$$

Ghost Variable Unification. We follow the same mechanism than the one defined in the WEAVE algorithm. A user-defined *unification* function $v : \mathcal{V}^* \rightarrow \mathbb{B}$ allows designer to identify variables that need to be unified while merging the fragment set.

Let $V_{in} = \{v \mid \exists f \in F, v \in inputs(hook(f))\}$, $V_{out} = \{v \mid \exists f \in F, v \in outputs(hook(f))\}$

$\rightsquigarrow () \stackrel{\perp}{\leftarrow} \{u(V, gensym(V, ctx), f') \mid \exists V \subseteq V_{in}, v(V)\} \stackrel{\perp}{\leftarrow} \{u(V, gensym(V, ctx), f') \mid \exists V \subseteq V_{out}, v(V)\}$

8.3.2 Illustrating Example

In this section, we consider three fragments: *bef* (FIG. 8.2(a)), *par* (FIG. 8.2(b)) and *aft* (FIG. 8.2(c)), and the following weave directive set:

$$\Omega = \{\dots, \omega(bef, B, \equiv), \dots, \omega(par, B, \equiv), \dots, \omega(aft, B, \equiv), \dots\} \in Directive^*$$

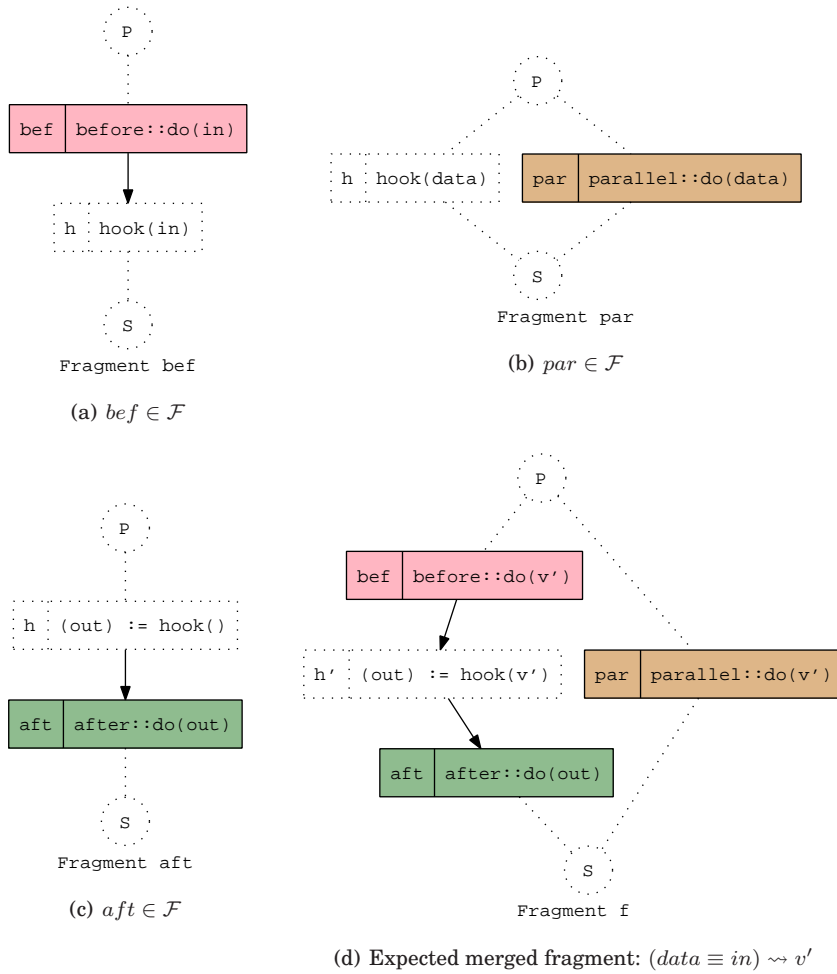


Figure 8.2: Illustrating the MERGE algorithm: $MERGE(\{bef, par, aft\}, \equiv, f)$

We use this example to illustrate the principles of the algorithm. The expected result (obtained after the execution of the generated actions) is depicted in FIG. 8.2(d).

- *Shared Join Point Identification.* According to the previously defined *shared* function, the composition engine identifies that B is a shared join point: $shared(B, \Omega) = \{aft, bef, par\}$. This information is raised to the designers, who now know that a shared join point exists in their system.

- *Fragment Activities Unification.* To create the *merged* fragment, three shared fragments are *discharged* into a newly created one (empty, denoted as f). Then, predecessors, successors and hooks elements are respectively unified, to build the final behavior.
- *Ghost Variable Unification.* The designers involved in the definition of *bef*, *aft* and *par* are informed of the situation. Consequently, they can assess their own fragment according to the other ones. In this example, the *data* variable used in the *par* fragment represents the same concrete element than the *in* variable defined in *bef*. These two variables are then unified into a new one (v').

8.3.3 Algorithm Description & Properties

The MERGE algorithm is fully described in ALG. 5. It takes as input a multiset of fragments (Φ), an unification function (v) and the expected name for the merged fragment (f). The first line initializes a *context*, and then fills the generated action list with initial actions: (i) creating an empty fragment named f , and then discharging all the fragments contained in Φ into f . The “fragment activity unification” step is realized through the lines three and four, which compute the set of activities to unify (P , S and H) and then generate the associated unification action. According to the v function, lines six and seven realize the “ghost variable unification” step. Finally, the generated action set is returned at line eight.

Algorithm 5 MERGE: $\Phi \times v \times f \mapsto actions$

Require: $\Phi = \{f_1, \dots, f_n\} \in \mathcal{F}^*$, $(v : \mathcal{V}^* \rightarrow \mathbb{B}) \in Function$, $f \in GroundTerm$, $\nexists \pi \in \mathcal{P}, name(\pi) = f$

Ensure: $acts \in Actions_{\leq}^*$

```

1:  $ctx \leftarrow context()$ ,  $acts \leftarrow (add_p(f, \emptyset, \emptyset, \emptyset)) \cup \{d(\phi, f) \mid \phi \in \Phi\}$  { Initialization }
2: { Fragment Activities Unification }
3:  $P \leftarrow \{\pi \mid \phi \in \Phi, \pi = \mathbb{P}(\phi)\}$ ,  $S \leftarrow \{\sigma \mid \phi \in \Phi, \sigma = \mathbb{S}(\phi)\}$ ,  $H \leftarrow \{h \mid \phi \in \Phi, h = hook(\phi)\}$ ,
4:  $acts \stackrel{\pm}{\leftarrow} (u(P, gensym(P, ctx), f), u(S, gensym(S, ctx), f), u(H, gensym(H, ctx), f))$ 
5: { Ghost Variables Unification }
6:  $V_{in} \leftarrow \{v \mid \exists \phi \in \Phi, v \in inputs(hook(\phi))\}$ ,  $V_{out} \leftarrow \{v \mid \exists \phi \in \Phi, v \in outputs(hook(\phi))\}$ 
7:  $acts \stackrel{\pm}{\leftarrow} \{u(V, gensym(V), f) \mid \exists V \subseteq V_{in}, v(V)\} \cup \{u(V, gensym(V), f) \mid \exists V \subseteq V_{out}, v(V)\}$ 
8: return  $acts$  { Exit }
```

The algorithm ensures the two following properties: (i) determinism and (ii) parallel composition.

- (i) *Determinism.* By construction, the algorithm generates unification actions according to the received set of fragments Φ , and the unification function v . Assuming that v is a deterministic function, the MERGE algorithm ensures to always compute the same actions set.
- (ii) *Parallel Composition.* The MERGE algorithm builds the final fragment by discharging all shared fragments into the merged one. The performed unifications never create direct relations between activities from different fragments. The *hook* activity is considered as a synchronization point for all the fragments: a post-hook activity a from a fragment f may transitively wait the end of an activity a' defined in another fragment f' . Considering the fact that these two behavior *share* the hook activity, such a wait is considered as expected. Consequently, the MERGE algorithm ensures that the behaviors defined in different fragments are executed in parallel **and** synchronized on the unified hook.
- (iii) *WEAVE Equivalence.* According to the parallel composition property, using a MERGE allows the highlighting of a shared join point, but does not interfere with the final composition result (obtained with multiple execution of the WEAVE algorithm). Assuming that designers does not modify the merged result, using in a WEAVE invocation (i) a directive consuming the merged fragment or (ii) the initial directives are equivalent. This behavior is expected: without any choice introduced by the designers, the MERGE algorithm must not interfere with the WEAVE one. Its intrinsic goal is to highlight shared behavior and support the oblivious assessment of merged fragments. The merged fragment is analyzed by the interference detection rules to highlight problems.

8.4 Taming Process Design Complexity

The previous section describes the MERGE algorithm *in-the-small*. We propose now a modeling approach which integrates together the WEAVE and the MERGE algorithm, as sketched in FIG. 8.1. A preliminary version of this approach is described in [Mosser et al., 2008a]. We use this approach and the associated implementation in the two case studies used to assess ADORE (PART IV).

8.4.1 Prerequisite: An Algorithm Invocation Pool (\mathbb{I})

The goal of this approach is to support designers while they design large processes. We consider now that several designers are working on the same system. As we describe in the previous chapter, they design *orchestrations*, *fragments*, and express WEAVE directives to integrate fragments into other processes.

We reify the invocations of the WEAVE algorithm as first class entities. Designers deposit their invocations in an invocation pool \mathbb{I} , in an oblivious way. When the system is asked to perform the composition, the invocations stored in the pool are executed, and the final system is built.

8.4.2 Principle: Automated Reasoning on the Invocation Pool

The principle of the proposed approach is simple: based on the invocations stored in \mathbb{I} , we automate the combination of interacting directives. This automation is done through an oracle named “*algorithm scheduler*”. It supports designers while composing processes at two levels: (i) a static analysis of the expressed directives and (ii) by scheduling the execution of the algorithm to reach designers goals.

Static Reasoning. This part of the reasoning is done statically, based on the element defined in the invocation pool. These mechanisms follow the same ideas than the ones defined in MATA [Whittle et al., 2009], but rely on a business-rules approach instead of a critical-pair analysis.

- *Invocation Union.* According to the WEAVE semantic, the directives defined in an invocation do not interfere with each others. Based on this property, the scheduler rewrites several WEAVE invocations targeting the same process as a single invocation using the union of the preexisting directives.

$$\begin{aligned} \text{Let } I_1 = \text{WEAVE}(\{\omega(f, B, \beta)\}, p) \in \mathbb{I}, I_2 = \text{WEAVE}(\{\omega(f', B', \beta')\}, p) \in \mathbb{I} \\ \rightsquigarrow \text{WEAVE}(\{\omega(f, B, \beta)\}, p, \omega(f', B', \beta')\}, p) \in \mathbb{I}, I_1 \notin \mathbb{I}, I_2 \notin \mathbb{I} \end{aligned}$$

- *Fragment Multiple Uses.* The algorithm *consumes* their input elements. As a consequence, if one decides to use the same artifact several times, it needs to be cloned before. The scheduler detects such multiple usage of the same fragment, expresses clone directives, and rewrites the invocation to use the clones instead of the original artifact.

$$\begin{aligned} \text{Let } I_1 = \text{WEAVE}(\{\omega(f, B, \beta)\}, p) \in \mathbb{I}, I_2 = \text{WEAVE}(\{\omega(f, B', \beta')\}, p') \in \mathbb{I} \\ \rightsquigarrow \text{CLONEPROCESS}(f, f') \in \mathbb{I}, \text{WEAVE}(\{\omega(f', B', \beta')\}, p') \in \mathbb{I}, I_2 \notin \mathbb{I} \end{aligned}$$

- *Shared join Point Merge.* When a shared join point is identified, the scheduler builds the associated MERGE², and uses a single WEAVE invocation to perform the integration. This situation is considered as potentially harmful (interferences may spawn), so the designers are informed with a warning. They can then assess the merged fragment in isolation with the other elements, and tackle the issue created by this shared join point.

$$\begin{aligned} \text{Let } I_1 = \text{WEAVE}(\{\omega(f, B, \beta_1)\}, p) \in \mathbb{I}, I_2 = \text{WEAVE}(\{\omega(f', B, \beta_2)\}, p) \in \mathbb{I} \\ \rightsquigarrow \text{MERGE}(\{f, f'\}, \beta_1 \cup \beta_2, \mu) \in \mathbb{I}, \text{WEAVE}(\{\omega(\mu, B, \beta)\}, p) \in \mathbb{I}, I_1 \notin \mathbb{I}, I_2 \notin \mathbb{I} \end{aligned}$$

²We do not address in ADORE the automatic composition of the binding functions $\{\beta_1, \dots, \beta_n\}$ into something more complicated than a function \cup_β which reaches the union of their image spaces

Execution Scheduling: Composition Chain. When invocations do not rely on each others, they can be executed in an arbitrary order. But when an invocation I uses as input an element created by an invocation I' (such as a cloned fragment), it is clear that I' must be executed before (i.e., the invocation is played to retrieve the associated actions set S , and S is executed to really create the needed element). The scheduler supports this chaining by analyzing dependencies³ in the expressed invocations, and executing the calls consequently.

8.4.3 Storing Designers Choices in a Knowledge Basis (Ξ)

The MERGE algorithm is used to assess the behavior built around a shared join point. If a merged fragment contains errors due to an unanticipated interference, the designer may express one or more actions to solve it. These actions must be capitalized to allow their reuse in similar situations.

Ξ , **the ADORE knowledge basis.** We define a very simple knowledge basis to store designers decisions. The basis can store *rules*, in a logical form $A \Rightarrow B$. If the prerequisite expressed as A is satisfied, the actions defined as B are concatenated with the ones generated by the usual algorithms.

Let μ a fragment obtained as the merge of two fragments f_1 and f_2 . An interference appears, and the designers decide to perform a re-ordering between two activities a and a' to avoid such a situation. The following rules is added into the knowledge basis:


$$\Xi \stackrel{\leftarrow}{\vdash} \text{merge}(\{f, f'\}) \Rightarrow a \prec a'$$

We consider now a later situation, where fragments f , f' and f'' must be woven on a shared join point. A merge is triggered, asking the ADORE engine to merge these three fragments. The previously added rule is satisfied (we are merging f and f'), and the associated action⁴ is added at the end of the generated ones by the scheduler:

$$S = \text{MERGE}(\{f, f', f''\}, \Xi, \phi) \stackrel{\leftarrow}{\vdash} \underbrace{\text{add}_r(a \prec a', \phi)}_{\Xi \text{ rule}}$$

Limitations & Perspectives. We consider here a syntactic solution, where designer must solve interferences at the activity instance level. At the technical level, the knowledge basis interrogation is not efficient: it must take care of activity derivation while satisfying the rules. We do not allow the expression of *business-driven* rules (e.g., “an invocation of the operation *crypt* must always be the first one”). The usage of semantic web technologies to tackle this limitation is exposed as a perspective. Another limitation is the absence of consistency verification in the knowledge basis. We assume that the rules expressed in Ξ are consistent regarding each others, and will not generate unexpected behavior.

8.5 Modeling PICWEB ...

In this section, we show how the previously described modeling process is used to build the PICWEB system. We use the “” symbol to denote an element expressed by the designers while they perform the composition. We use the u_0 symbol to denote the initial universe.

8.5.1 The Initial PICWEB System: FLICKR®, PICASA™ & Timeout

(u_0) Base Universe. We consider here the realization of the requirements as business processes. The base universe contains:

³We assume that circular dependencies were previously rejected or manually handled.

⁴At the implementation level, we must perform an analysis of the *cloned* entity to retrieve the current element instead of their origins while handling cloned processes.

- ✦ Two orchestrations *getPictures* (FIG. 5.7(a)) to retrieve the set of relevant pictures and *truncate* (FIG. 4.7) to restrict a set cardinality up to a given threshold.
- ✦ Two fragments: *addPicasa* (FIG. 7.11) to retrieve pictures stored in the PICASA™ repository, *timeout* (FIG. 7.12) to handle an eventual repository timeout.

(u_1) **Introducing PICASA™ & Timeout.** The *addPicasa* fragment is given by the user, and asked to be woven on the FLICKR® invocation. According to the requirements, the timeout concern impacts both FLICKR® and PICASA™ repository. We factorized this concern into a single fragment *timeout*. This fragment needs to be woven into the *addPicasa* fragment (on the PICASA™ invocation, aP_1) and into the *getPictures* orchestration (on the FLICKR® invocation, a_2). Designers express the two following WEAVE invocations I_1 and I_2 to perform such a goal:

$$\begin{aligned} \text{✦ } I_1 &= \text{WEAVE}(\{\omega(\text{timeout}, \{aP_1\}, \equiv)\}, \text{addPicasa}) \\ \text{✦ } I_2 &= \text{WEAVE}(\{\omega(\text{addPicasa}, \{a_2\}, \equiv), \omega(\text{timeout}, \{a_2\}, \equiv)\}, \text{getPictures}) \end{aligned}$$

Based on these two invocations, the algorithm scheduler identifies the following situations: (i) the *timeout* fragment is used twice, (ii) the a_2 activity of the *getPictures* process is a shared join point and finally (iii) the second invocation I_2 uses a fragment enhanced by the invocation of I_1 . The scheduler performs the re-ordering, and executes the following sequence of actions:

$$\begin{aligned} \tau_1 &= do^+(\text{CLONEPROCESS}(\text{timeout}, t'), u_0) \\ \tau_2 &= do^+(\text{WEAVE}(\{\omega(\text{timeout}, \{aP_1\}, \equiv)\}, \text{addPicasa}), \tau_1) \\ \tau_3 &= do^+(\text{MERGE}(\{\text{addPicasa}, \text{timeout}\}, \equiv, \mu), \tau_2) \\ \tau_4 &= do^+(\text{WEAVE}(\{\omega(\mu, \{a_2\}, \equiv)\}, \text{getPictures}), \tau_3) \end{aligned}$$

The shared join point information is sent to the designers, who examine the merged fragment (denoted as μ) depicted in FIG. 8.3(a). The interference detection mechanism identifies a concurrent access to a variable in the merged process: v^* is concurrently accessed by t_2 and aP_2 ⁵. An equivalence between the two stopwatch activities (t_1 and t'_1) is also detected. Designers decide (i) that the activity used to merge the picture sets must be the final one (aP_2) and (ii) that the two stopwatch must be unified. Knowledge is added in the knowledge basis Ξ , to store this information, and the algorithm is re-played to build the expected result according to these new information (FIG. 8.3(b)).

$$\begin{aligned} \text{✦ } \Xi &\stackrel{+}{\leftarrow} \text{merge}(\{\text{addPicasa}, \text{timeout}\}) \Rightarrow t_3 \prec aP_2 \wedge u(\{t_1, t'_1\}, t) \\ \text{Acts} &= \text{MERGE}(\{\text{addPicasa}, \text{timeout}\}, \equiv, \mu) \\ \text{Acts} &\stackrel{+}{\leftarrow} (\text{add}_r(t_3 \prec aP_2, \mu), u(t_1, t'_1, \mu)) \quad (\text{cf. } \Xi) \\ \tau'_3 &= do^+(\text{Acts}, \tau_2) \\ u_1 &= do^+(\text{WEAVE}(\{\omega(\mu, \{a_2\}, \equiv)\}, \text{getPictures}), \tau'_3) \end{aligned}$$

8.5.2 Introducing Fault Management

We consider now a version of PICWEB which needs to deal with FLICKR® faults. Based on the requirements of the PICWEB system, we consider that the FLICKR® repository may respond fault messages. In this case study, we consider two different faults: (i) the given *key* is not valid because of an account deactivation (**invalid**) and (ii) the daily call quota is reached (**overflow**). The requirements ask to silently ignore the fault

- ✦ We use the two fragments⁶ depicted in FIG. 8.4 to handle FLICKR® faults.

⁵At the implementation level, the engine also identifies a concurrent access between h' and t_2 . The activity exclusiveness function does not take into account downstream *weakWait* relations. However, one can flag this conflict as a *false-positive* one.

⁶These two fragments are instantiation of a *generic* one, defined to handle a fault f . We describe in APP. B an *instantiation* algorithm (based on the CLONEPROCESS algorithm, including user-given substitution while cloning) used to perform such a task.

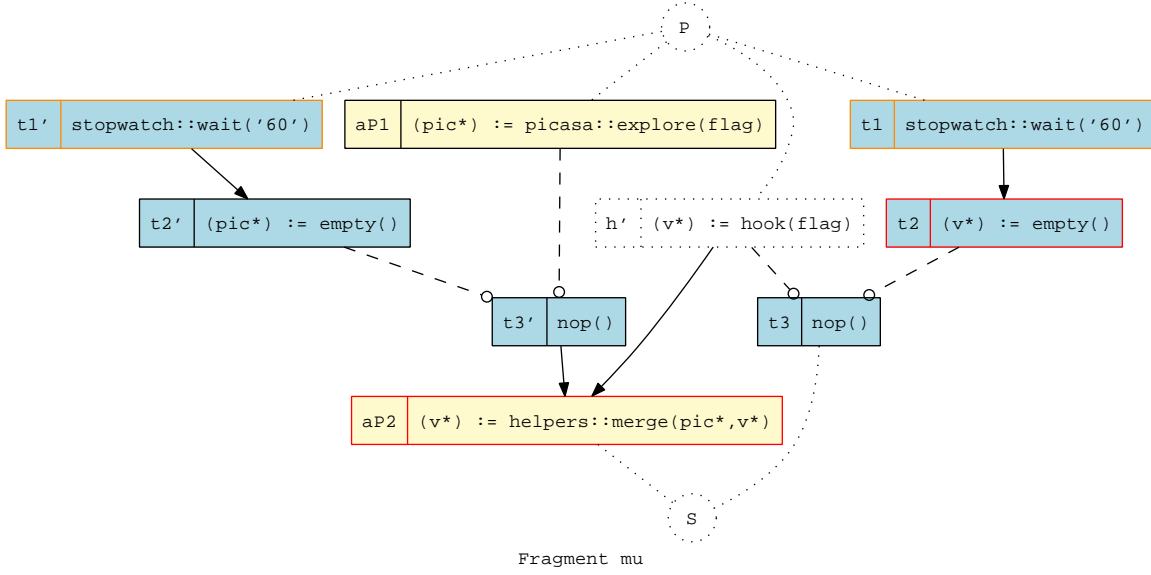
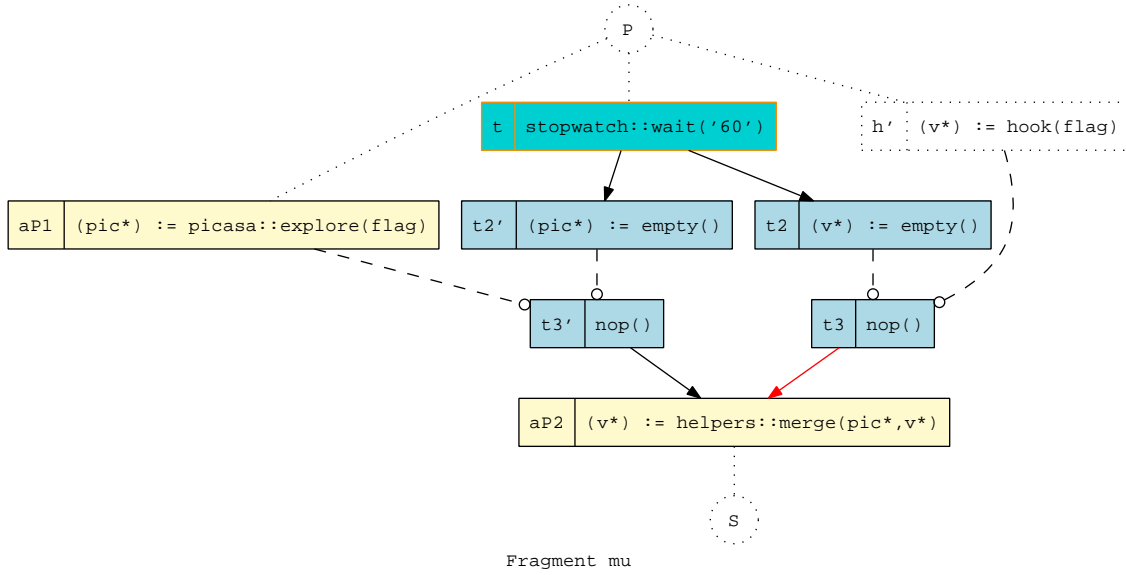
(a) Automatically built $\mu \in \text{procs}(\tau_3)$: interference on v^* for $\{aP_2, t_2\}$, $t_1 \equiv t'_1$ (b) $\mu \in \text{procs}(\tau'_3)$, with Ξ actions (re-ordering & unification)

Figure 8.3: PICWEB merged fragment: PICASA™ & timeout concerns

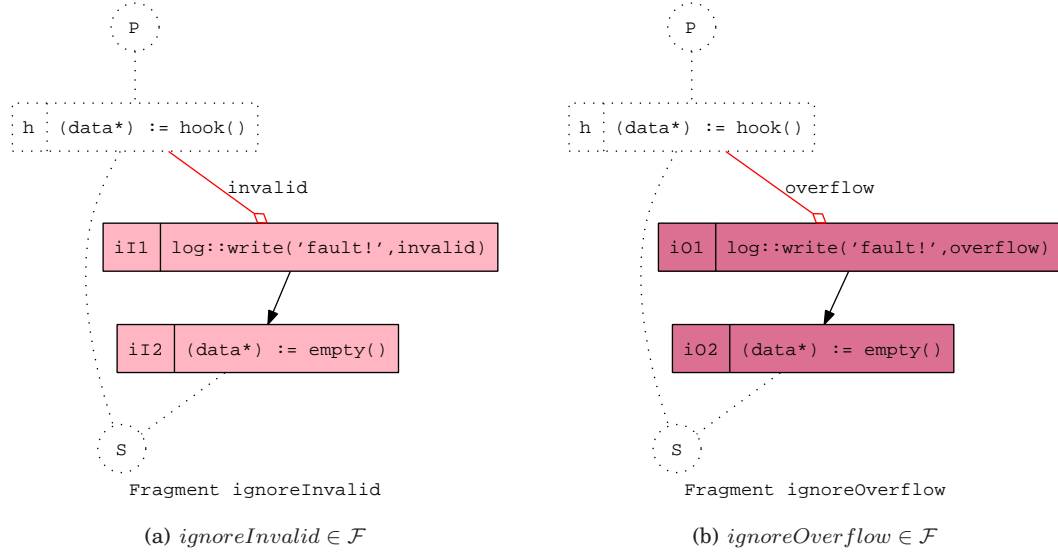


Figure 8.4: Fault management (silent ignore) fragments defined in PICWEB

These two fragments must be woven on the **FLICKR**[®] invocation defined in the **getPictures** business process. Designers define the following **WEAVE** invocation to fulfill this requirement:

$$\clubsuit I_3 = \text{WEAVE}(\{\omega(ignoreInvalid, \{a_2\}, \equiv), \omega(ignoreOverflow, \{a_2\}, \equiv)\}, getPictures)$$

While reasoning on I_1 , I_2 and I_3 , the algorithm scheduler identifies that the a_2 shared join point involves now four fragments: *addPicasa*, *ignoreInvalid*, *ignoreOverflow* and *timeout*. The algorithms calls are rewritten as the following:

$$\begin{aligned} Acts &= \text{MERGE}(\{addPicasa, ignoreInvalid, ignoreOverflow, timeout\}, \equiv, \mu) \\ Acts &\stackrel{+}{\leftarrow} (add_r(t_3 \prec aP_2, \mu), u(t_1, t'_1, \mu)) \quad (\text{cf. } \Xi) \\ \tau_5 &= do^+(Acts, \tau_2) \\ \tau_6 &= do^+(\text{WEAVE}(\{\omega(\mu, \{a_2\}, \equiv)\}, getPictures), \tau_5) \end{aligned}$$

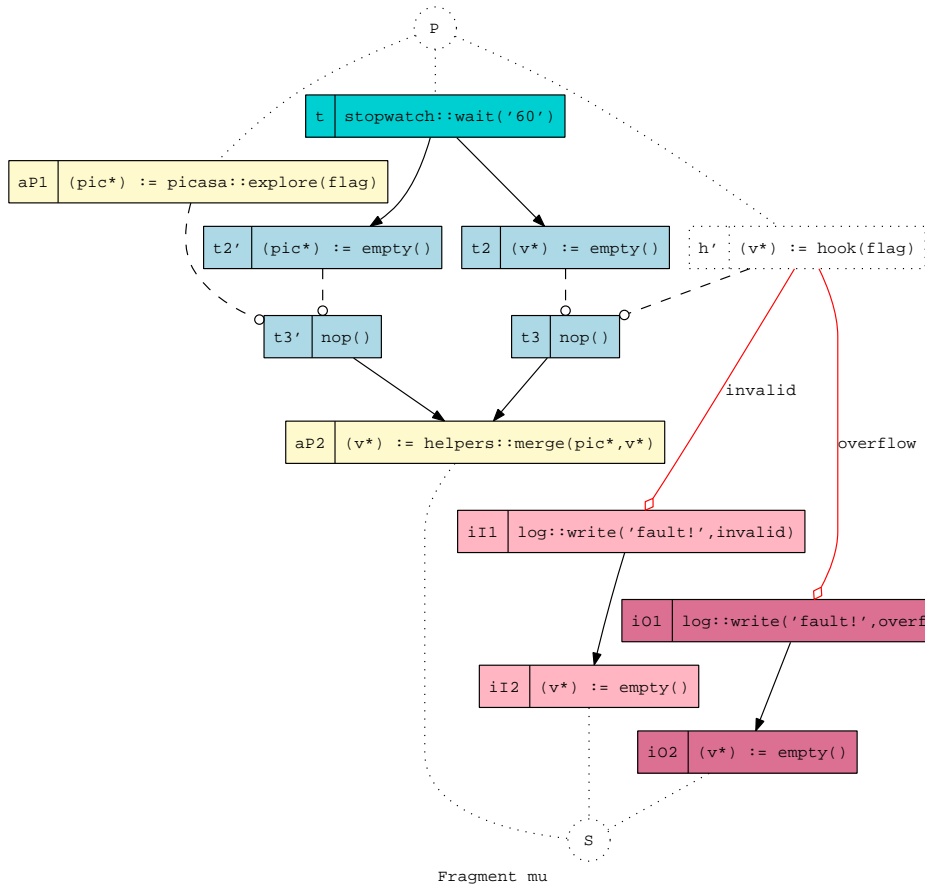
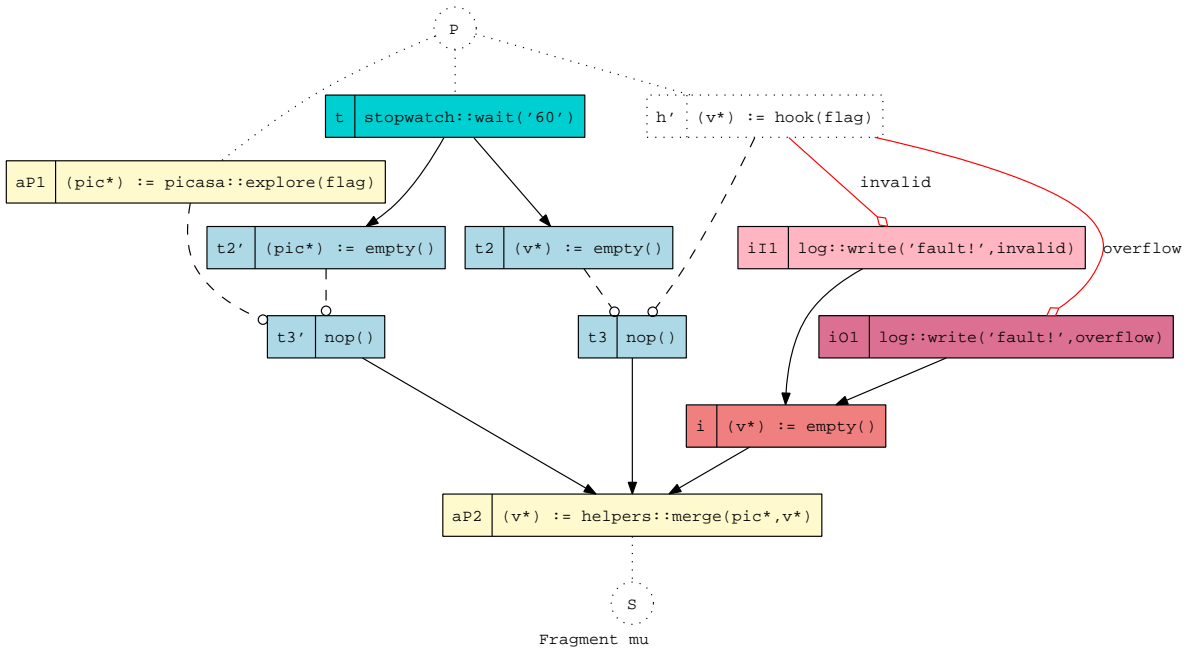
While assessing the μ fragment, the designers identify two issues: (i) the aP_2 activity must be the last one (a concurrent access on v^* is identified by the engine) and (ii) the two activities used to ignore a fault (iI_2 and iO_2) may be unified into a single one (considering that $iI_2 \equiv iO_2$). These decisions are added in the knowledge repository, and the algorithm produces the expected result as output.

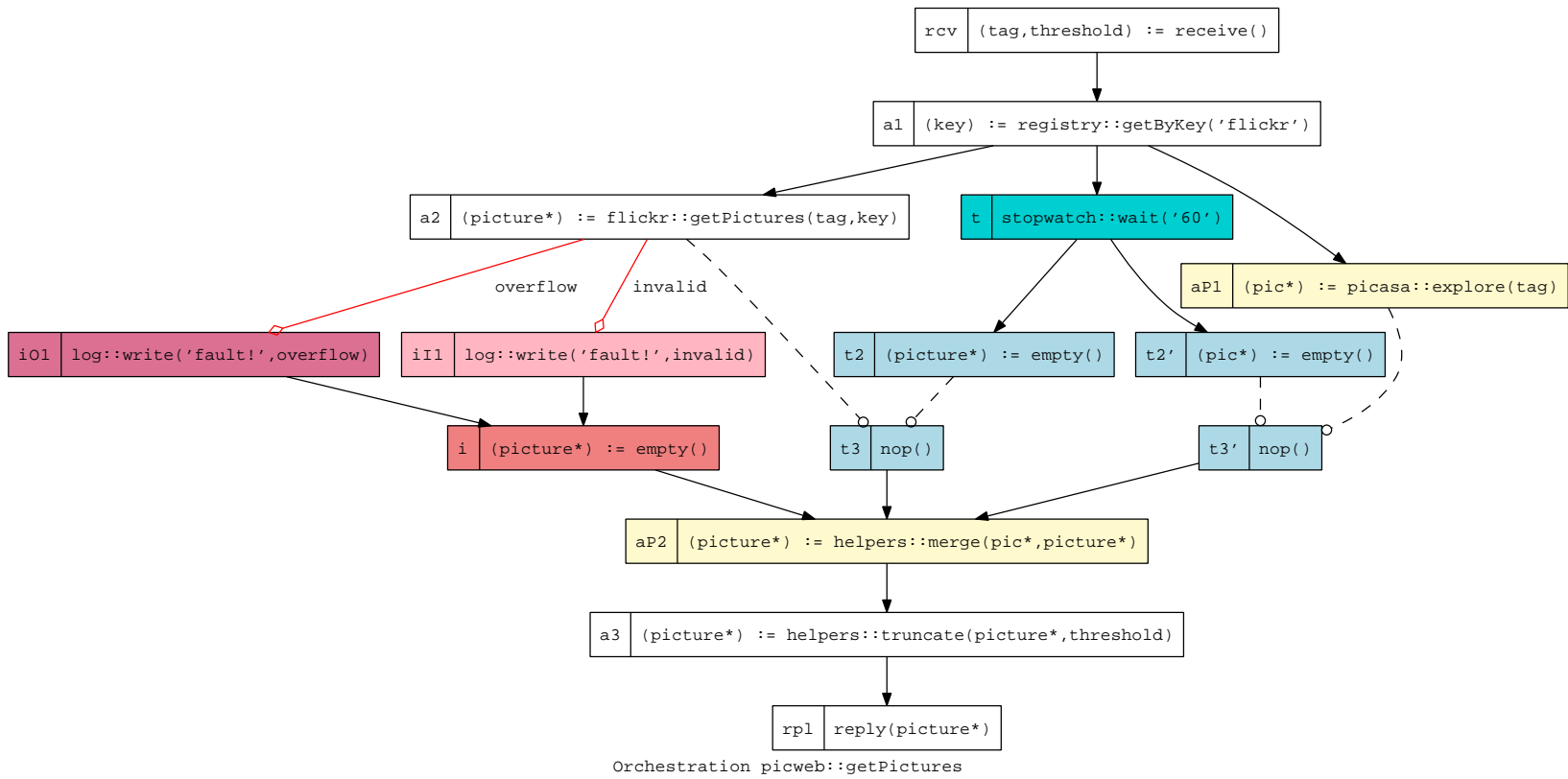
$$\begin{aligned} \clubsuit \Xi &\stackrel{+}{\leftarrow} merge(\{addPicasa, ignoreInvalid\}) \Rightarrow iI_1 \prec aP_2 \\ \clubsuit \Xi &\stackrel{+}{\leftarrow} merge(\{addPicasa, ignoreOverflow\}) \Rightarrow iO_1 \prec aP_2 \\ \clubsuit \Xi &\stackrel{+}{\leftarrow} merge(\{ignoreInvalid, ignoreDisconnection\}) \Rightarrow u(\{iI_1, iO_1\}, i) \\ Acts &= \text{MERGE}(\{addPicasa, ignoreInvalid, ignoreOverflow, timeout\}, \equiv, \mu) \\ Acts &\stackrel{+}{\leftarrow} (add_r(t_3 \prec aP_2, \mu), u(t_1, t'_1, \mu), add_r) \\ Acts &\stackrel{+}{\leftarrow} (add_r(iI_1 \prec aP_2, \mu), add_r(iO_1 \prec aP_2, \mu), u(\{iI_1, iO_1\}, i, \mu)) \\ \tau'_5 &= do^+(Acts, \tau_2) \\ u_2 &= do^+(\text{WEAVE}(\{\omega(\mu, \{a_2\}, \equiv)\}, getPictures), \tau'_5) \end{aligned}$$

We represent in FIG. 8.6 the *getPictures* process obtained in u_2 .

8.5.3 Step-Wise Development: Adding Cache Mechanisms

We show in the previous section how **ADORE** is used to support the modeling of large processes. With this approach, we consider that a set of composition algorithm invocations are simultane-

(a) μ , automatically built(b) μ , including actions added in the knowledge basisFigure 8.5: PICWEB merged fragment μ (including PICASA™ & timeout concerns)

Figure 8.6: *getPictures* business process, with timeout, PICASA™ & fault handling concerns

ously defined, and executed independently of the order they were expressed. In usual step-wise development methods, features are composed one after one, in a sequential way. We defend that the approach proposed with ADORE is complementary: one can use ADORE to implement a step-wise development method, as we sketched in the motivations section.

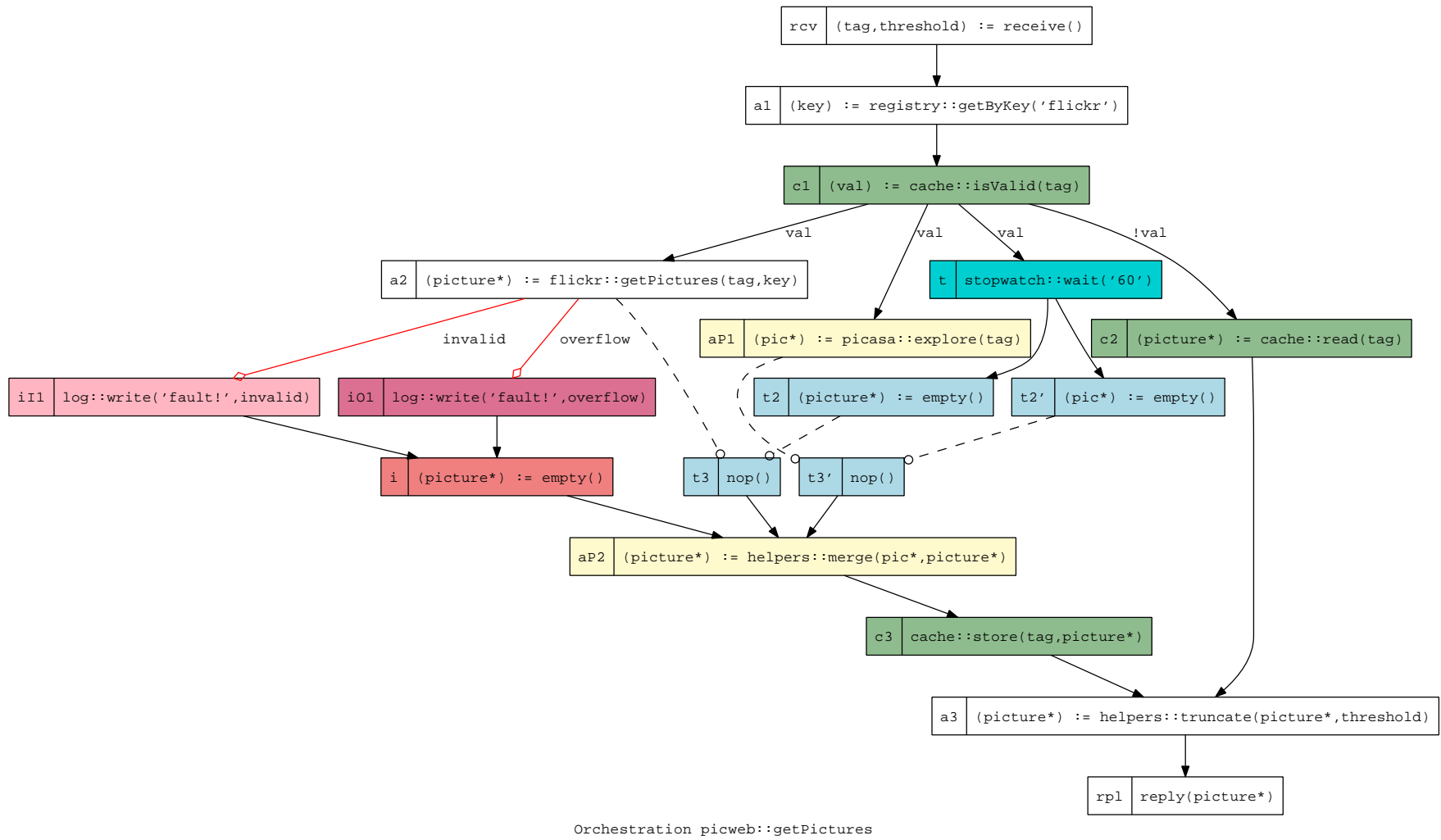
PICWEB Illustration. We illustrate this idea on the PICWEB example. We consider that the system obtained in the previous section is the result of a first composition step. A designer wants to add a cache mechanism to end the design of the *getPictures* orchestration. This is clearly the **next** step according to PICWEB semantic:

- Step 0 was the implementation of the base processes,
- Step 1 dealt with the enhancement of the base process to deal with multiple repositories and the associated faults,
- Step 2 will focus now on processes performance.

As a consequence, the cache must store the pictures retrieved from the two repositories. At the ADORE level, the *cache* fragment FIG. 5.7(b) is woven on all the activities defined in the composed *getPictures*, excepting $\{rcv, a_1, a_3, rpl\}$.

$$\begin{aligned} \Leftarrow S &= \text{WEAVE}(\{\omega(\text{cache}, \text{acts}(\text{getPictures}) \setminus \{rcv, a_1, a_3, rpl\}, \equiv), \text{getPictures}\}) \\ \Leftarrow u_3 &= do^+(S, u_2) \end{aligned}$$

The final process, obtained after this new composition step, is depicted in FIG. 8.7.

Figure 8.7: Complete version of the PICWEB *getPictures* business process

Improving Performances with the INLINE Algorithm

«You cannot simply put something new into a place. You have to absorb what you see around you, what exists on the land, and then use that knowledge along with contemporary thinking to interpret what you see.»

Tadao Ando

Contents

9.1 Introduction	123
9.2 Motivating Example: PICWEB Slow Execution	124
9.2.1 Implementation Benchmark: Average Invocation Time	124
9.2.2 Weights & Costs: “Let’s reason about messages ...”	125
9.2.3 Inlined Benchmark	126
9.3 Definition of the INLINE algorithm	127
9.3.1 Principles	127
9.3.2 Description	128
9.3.3 Intrinsic Limitations: Inlining is not a “Holy Grail”	130
9.3.4 Integration in the ADORE Development Approach	130
9.4 Using the INLINE algorithm on PICWEB	130

9.1 Introduction

In the previous section, we described the implementation of *Separation of Concerns* techniques such as the WEAVE and the MERGE algorithm. In this section, we focus on a new kind of composition, inspired by the compilation technological domain: “process inlining”. We understand the word *inline* according to its definition in the C++ language standard:

*“The **inline** specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism.”* [ISO, 2003]

In this chapter, we start by motivating the need for such a composition in the context of SOA. We describe in SEC. 9.2 how an implementation of the PICWEB system can be speeded up using such a technique. Then, we provide in SEC. 9.3 a formal description of the algorithm, and its intrinsic limitation. We finally end this chapter in SEC. 9.4 by showing how the algorithm is used to automatically enhance the PICWEB models.

9.2 Motivating Example: PICWEB Slow Execution

We identified the need for an `INLINE` algorithm with the usage of the `JSEDUITE` instance daily used in the `POLYTECH’NICE` engineering school. The fact was simple: the `JSEDUITE` system was slow. Very slow. This section restricts the problem to its pure essence, and expresses it on the `PICWEB` example.

Technical Information. All the experimentations made in this section were made on a Mac-Book Pro computer, running Mac OS X 10.4.11 and built upon a 2.2 GHz Intel Core 2 Duo processor and 2Go of RAM memory. We use the Glassfish 2.1 application server, and the associated BPEL engine. The client is written using JAX-WS web services stack, as included in Netbeans 6.5.1. To control the execution time of the `FLICKR`[®] service and avoid network latency or uncontrolled server overload, we use a simulator for this service. This simulator generates a set of fake picture URLs instead of exploring the real repository.

9.2.1 Implementation Benchmark: Average Invocation Time

Based on an intensive usage of `PICWEB` processes implementation (APP. D), we remark that the *getPictures* process seems to be slow. To ease the description of the problem, we consider here the initial version of `PICWEB`. It contains two business processes: (i) *truncate* to restrict set cardinality up to a given threshold, and (ii) *getPictures* to retrieve the set of pictures expected by the final user. We also consider the `FLICKR`[®] service, used to explore the `FLICKR`[®] repository content. A *registry* is needed to use `FLICKR`[®] at a technical level, but it is ignored in the context of this experience since it produces results equivalent to the `FLICKR`[®] service.

To assess the system, we instrumented the `PICWEB` implementation to control: (i) the average execution time of a service invocation and (ii) the cardinality of the picture set handled by the processes. We measure the invocation time on the client-side¹, in milliseconds. To avoid server-side garbage collection slowdown, we consider an average execution time metric (based on hundred invocations). We depict in FIG. 9.1 the result of this benchmark. The *x* axis represents the controlled set cardinality used for this experience. The *y* axis represents the average execution time for each benchmarked service.

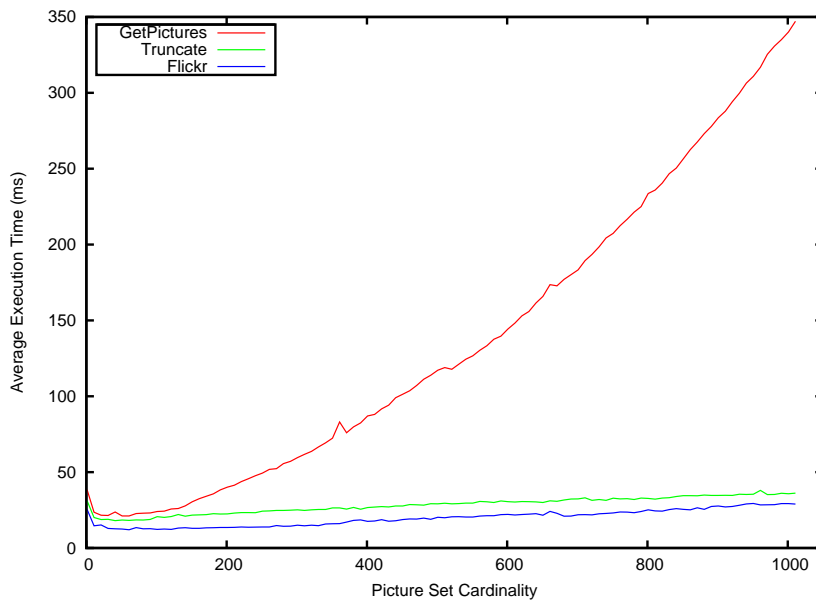


Figure 9.1: Average exec. time (ms) associated to *getPictures*, *truncate* and *flickr* invocations

¹We consider the network latency as a constant for this experience, as we host client and server on the same physical machine.

Data analysis. The average execution of the FLICKR® and the *truncate* operations are “constants”. They follow a linear progression while the cardinality is increased, and stay below an average response time of 40ms when using a set containing a thousand elements. Considering that *truncate* is implemented as a BPEL orchestration and FLICKR® as a JAX-WS, we assume that such technological choices do not impact the performances in an effective way in this context. The *getPictures* operation follows a completely different scheme, and takes up to 340ms to answer while handling a set of a thousand pictures. Considering that the two other operations response times follow a regular progression, it seems that the internal logic of the *getPictures* operation impacts its performance.

Assumption: “Intensive Message Exchanges Generates Slowdown”. The *getPictures* orchestration differs from the *truncate* one by the number of exchanged messages. The *truncate* behavior is mostly internal, and the orchestration does not exchange data with the external world. The *getPictures* orchestration follows an opposite scheme, coordinating several message exchanges between the services involved in the PICWEB system.

9.2.2 Weights & Costs: “Let’s reason about messages ...”

In this section, we use a very simple mathematical representation of the exchanged messages to assess the previous assumption. We base our model on two notions: (i) the *weight* of the exchanged elements and (ii) the *cost* of an operation invocation (expressed in terms of exchanged data weights). Even if this model does not take care of conditional branches or exceptional replies, it is sufficient to expose the problem tackled by the INLINE algorithm in simple terms.

Notations. We use the following notations to model message exchanges:

- *Weight*:
 - We denote as w_e the weight of a scalar element e . For simplification purpose, we consider that w_e does not depend on the element type.
 - Let s^* a set of scalar elements of same weight. Its weight is equals to $|s^*|w_e$.
 - An exchanged *message* is composed by several elements $e^* = \{e_1, \dots, e_n\}$ binded together with *glue* information. This glue as a constant weight w_g . The weight of a message is then defined as $|e^*|w_e + w_g$
- *Cost*:
 - An operation o receives an input message in_o , and replies an output message out_o . The consumption cost associated to an operation is defined as the sum of the weights associated to its input and output messages.
 - If the operation is defined as an orchestration, its cost includes the costs of all the invocation performed in the orchestration.

Instantiation in PICWEB context. Base on these notations, we express now the *cost* associated to the invocation of a PICWEB-defined operation. PICWEB artifacts are handling elements such as integers, string and URLs. As a consequence, we consider their weights as equivalent and constant

- FLICKR® receives a *tag* and a *key* as input, and returns a set of pictures p^* as output.

$$\rightsquigarrow Cost(flickr) = (w_g + 2w_e) + (w_g + |p^*|w_e) = 2w_g + (|p^*| + 2)w_e$$

- *truncate* receives as input a data set p^* and a *threshold*. It returns a set of known cardinality (*threshold*) as output. The orchestration does not call others services.

$$\rightsquigarrow Cost(truncate) = (w_g + w_e + |p^*|w_e) + (w_g + threshold.w_e) = 2w_g + (|p^*| + threshold + 1)w_e$$

- *getPictures* receives as input a *tag* and a *threshold*. It returns a set of pictures, restricted to *threshold*. The orchestration calls the *flickr* operation and the *truncate* one, added to its interface cost.

$$\rightsquigarrow \text{Cost}(\text{getPictures}) = (w_g + 2w_e) + (w_g + \text{threshold}.w_e) + \text{Cost}(\text{flickr}) + \text{Cost}(\text{truncate}) \\ = 6w_g + (5 + 2\text{threshold} + 2|p^*|)w_e$$

In the context of this experience, we are focusing on the number of exchanged messages. As a consequence, we consider as negligible the glue used to define messages: $w_g = 0$. We use an arbitrary size for each exchanged element: $w_e = 1$. To strengthen the difference between the existing picture set and the restricted one, we consider that the user asks for a threshold equal to 1. The costs associated to the PICWEB processes is then simplified as the following:

$$\text{Cost}(\text{flickr}) = |p^*| + 2, \text{Cost}(\text{truncate}) = |p^*| + 2, \text{Cost}(\text{getPictures}) = 2|p^*| + 7$$

Key Idea: Optimizing Performances with Inlining. Assuming the existence of an algorithm able to *inline* an orchestration o into another one o' , one can enhance the performances (*i.e.*, reduce its cost) of a process by *inlining* o into o' . As a consequence, the cost associated to o' will be decreased by the cost associated to o . We denote as *getPictures'* a version of the process where the *truncate* process was inlined. We represent in FIG. 9.2 these two linear functions.

$$\text{Cost}(\text{getPictures}') = \text{Cost}(\text{getPictures}) - \text{Cost}(\text{truncate}) = |p^*| + 5$$

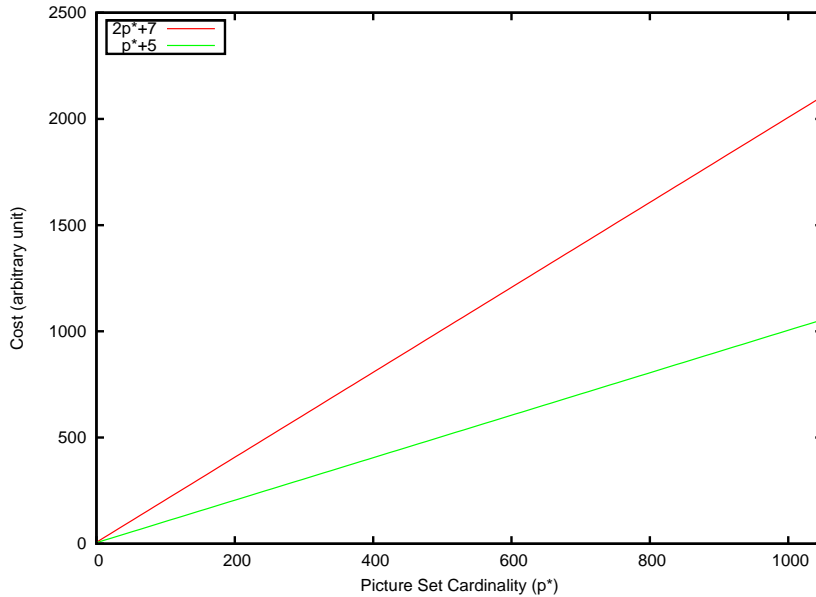


Figure 9.2: Cost Model instantiated on the *getPictures* process

9.2.3 Inlined Benchmark

To validate this model, we implement an inlined version of the *getPictures* process, and run the same benchmark. We depict in FIG. 9.3 the average execution times associated to such a comparison benchmark.

The cost model predicts a benefit lesser than the one empirically experimented. This situation is explained as the following:

- We made strong assumptions on the size of the data. For example, in a real-life usage of PICWEB, URLs are more heavy than simple integers. These assumptions allow to simplify

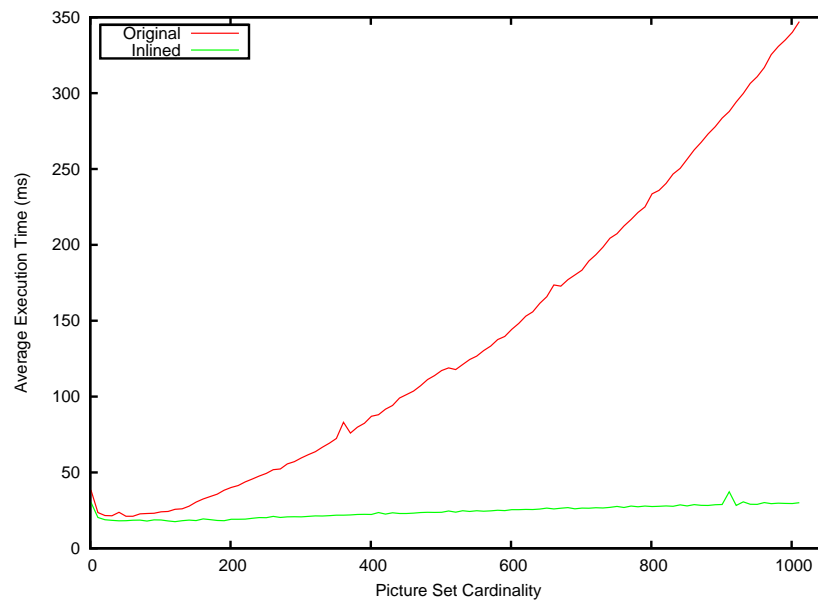


Figure 9.3: Comparing the inlined *getPictures* process performances and the original one

the model and focus on the number of exchanged messages. The other parameters may only change the y-intersect coefficient to tune the function.

- We do not model the intrinsic execution semantic and limitations associated to the BPEL engine available in the Glassfish application server². We only represent message exchanges, in an abstract way, independently of any application server or execution engine. This model is only used to support the intuitive benefits hidden in an inlining mechanism.

9.3 Definition of the INLINE algorithm

We present now the INLINE algorithm, defined to tackle the issue described in the previous section.

9.3.1 Principles

Expressiveness. We do not address in this work the use of compilation heuristics³ to automatically perform inlining in an efficient way [Chakrabarti and Liu, 2006]. The ADORE implementation of an inlining mechanism is inspired by the `inline` keyword defined in the C++ language [Stroustrup, 2000]. This keyword allows developers to explicitly ask the compiler to inline the content of the flagged function when called.

Implementation. Performing an *inlining* is similar to the WEAVE algorithm. The inlined process is discharged into the original one, and the relations defined in the original are rearranged to target the newly added activities. Inlined variables are substituted by original ones, according to a binding function β . The INLINE algorithm differs from the WEAVE one on three points:

- It is only possible to inline an *invocation*. Handling a block of activities or activities of other kinds makes no sense in this context.

²The observed averaged execution time function associated to the initial *getPictures* process looks like a quadratic function instead of a linear one.

³However, according to the experiments made with Glassfish, a simple heuristic should be “always inline”. This assumption needs to be verified empirically. Such an experiment is out of the scope of this work, as dedicated to a given BPEL execution engine. A discussion on the topic was started on the OpenESB users mailing list. The archive of the list contains leads to follow to setup such an experiment. <http://bit.ly/9X0cAT>

- The **INLINE** algorithm does not conserve several activities. The normal message exchanges activities (*i.e.*, with a kind equal to *receive* or *reply*) defined in the inlined process are deleted. In the original process, the invocation of the inlined process is consumed.
- Variables are replaced following an opposite method than the **WEAVE** one. While weaving, hook inputs are mapped with the variables used as input by the targeted activity, and hook outputs are mapped to the output variables of the targeted activity. While inlining, we need to substitute the variable received by the inlined orchestration (*i.e.*, used as output of the *receive* activity) by the ones *sent* to it by the original process (*i.e.*, used as input by the invocation). Respectively, variables used as input by the inlined *reply* are substituted by the variables used as output by the preexisting invocation.

Illustration. We illustrate the **INLINE** algorithm using a data-handling system, built upon two simple orchestrations: (i) *o* (FIG. 9.4(a)) and (ii) *partition* (FIG. 9.4(b)). The orchestration *o* receives an integer set raw^* , and a integer value x . It partitions raw^* according to x by invoking a dedicated operation (act_1). Then, the two retrieved sets pUp^* and $pDown^*$ are sent to a dispatch service (act_2). The retrieved data set res^* is then replied to the caller.

We depict in FIG. 9.4(c) the process expected after inlining the activity act_1 . We artificially color the activities to keep trace of their origin. The algorithm deletes the inlined invocation (act_1), and the interface of the *partition* process (p_1 and p_7). The variables defined in the *partition* process are substituted by variables defined in the *o* process, according to the following binding: $t \mapsto x$, $data^* \mapsto raw^*$, $up^* \mapsto pUp^*$, $down^* \mapsto pDown^*$.

Advantages. Based on the previously defined model, this operation avoid the exchange of two messages: (i) the *partition* input message ($w_g + (1 + |raw^*|)w_e$) and (ii) the *partition* output message ($w_g + (|raw^*|)w_e$). Consequently, we identify the two advantages at using the **INLINE** algorithm:

- At the conceptual level, the two processes are kept separated, and the *partition* orchestration is then factorized if multiply needed in a given system.
- At the deployment level, one can automatically inline processes and then enhance performances. In this particular case, we denote as $Cost(o')$ the cost associated to the inlined version of the orchestration *o*:

$$Cost(o') = Cost(o) - 2w_g + (1 + 2|raw^*|)w_e$$

9.3.2 Description

We provide in ALG. 6 the formal description of the **INLINE** algorithm. The algorithm takes as input an activity *act*, and the process to inline *inl*. Like the **WEAVE** algorithm, we use a β function to perform variable binding⁴. After an initialization step which generates the discharge of *inl* inside *o* (line 1), the algorithm starts by identifying and then generating the variables substitutions needs to adjust the data-flow (lines 3, 4). Then, it re-orders the predecessors and successors of *a* (line 6,7). Finally, it cleans up the process by deleting useless relations and activities (lines 9 \rightarrow 12).

Limitations. We made the choice to present in this document a limited version of the algorithm, to lighten its description⁵. The algorithm presented here is a simplification of the real one, which is more “technical”. For example, in this version, we do not take care of the different existing kinds of relations available in **ADORE**, and assume that there are always *waitFor* relations between the identified activities (lines 6 and 7). This version suffers from variable fragility, that is, the substitutions may lead to parallel write conflict when two inlined variables

⁴At the implementation level, this mapping is automatically done following the named parameter convention described in CHAP. 4.

⁵The source code of the complete **INLINE** algorithm is available on the **ADORE** website. See APP. B for details on algorithm implementations.

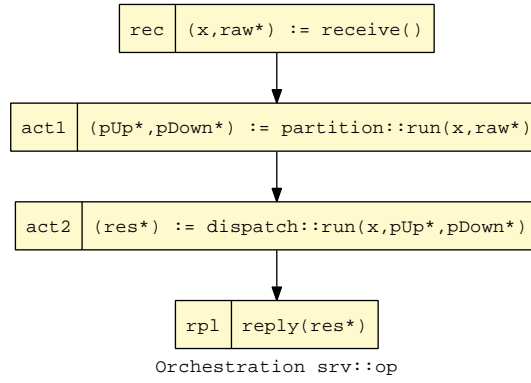
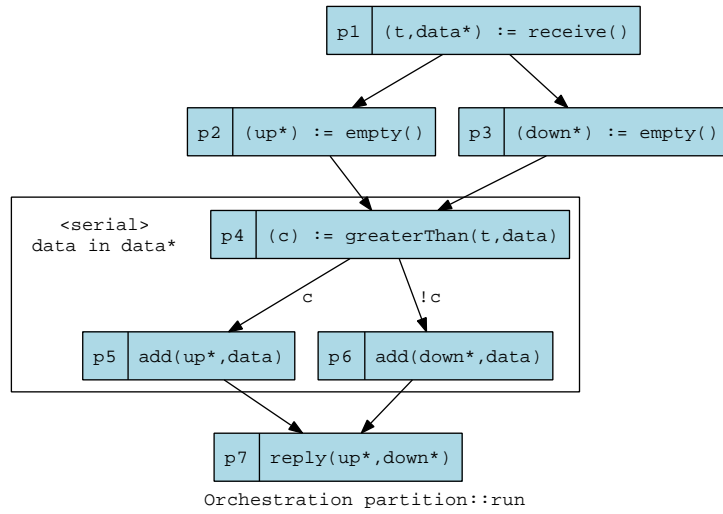
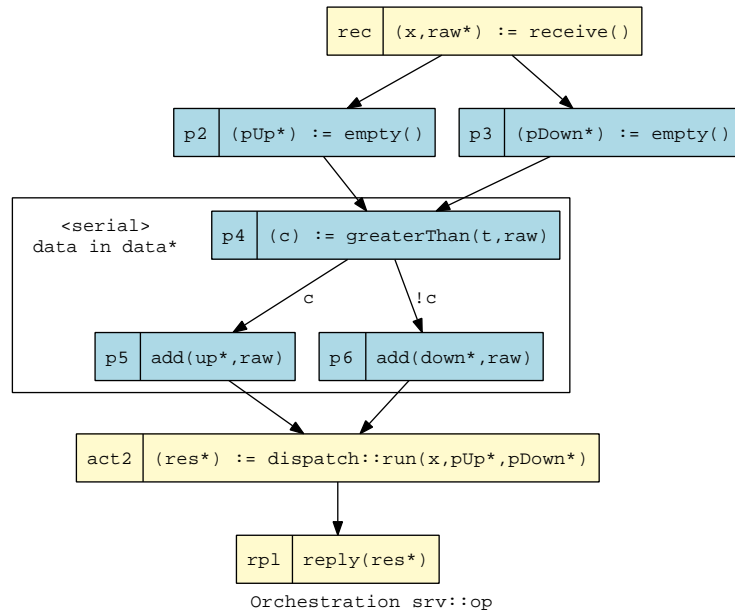
(a) Orchestration *o*: partition a data set and asks for a dispatch(b) Behavior used to define the *partition* business process(c) Orchestration *o* after the inlining of the activity *act₁*

Figure 9.4: Artifacts used to illustrate the INLINE algorithm

Algorithm 6 INLINE: $act \times inl \times \beta \mapsto r$

Require: $act \in \mathcal{A}$, $kind(a) = invoke(s, o)$, $inl \in \mathcal{O}$, $extra(inl) = orch(s, o)$, $(\beta : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{B}) \in Function$
Ensure: $r \in Actions_{\leq}^*$

```

1: Let  $p \in \mathcal{P}$ ,  $act \in acts(p)$ ,  $r \leftarrow (d(inl, p))$  { Initialization }
2: { Substituting Variables }
3:  $r \stackrel{\pm}{\leftarrow} \{r(v_{rec}, v_p, p) \mid \exists v_{rec} \in outputs(entry(inl)), \exists v_p \in vars(p), \beta(v_{rec}, v_p)\}$ 
4:  $r \stackrel{\pm}{\leftarrow} \{r(v_{rpl}, v_p, p) \mid \exists \alpha \in acts(inl), kind(\alpha) = reply, \exists v_{rpl} \in inputs(\alpha), \beta(v_{rpl}, v_p)\}$ 
5: { Re-Ordering Predecessors and Successors of  $a$  }
6:  $r \stackrel{\pm}{\leftarrow} \{add_r((\alpha, a_{inl}, l), p) \mid \exists (\alpha, a, l) \in rels(p), \exists entry(inl) \prec a_{inl} \in rels(inl)\}$ 
7:  $r \stackrel{\pm}{\leftarrow} \{add_r((a_{inl}, \alpha, l), p) \mid \exists a \prec \alpha \in rels(p), \exists r \in acts(inl), kind(r) = reply, \exists (a_{inl}, r, l) \in rels(inl)\}$ 
8: { Deleting Activities and Relations }
9:  $r \stackrel{\pm}{\leftarrow} \{del_r((\alpha, a, l), p) \mid \exists (\alpha, a, l) \in rels(p)\} \cup \{del_r((a, \alpha, l), p) \mid \exists (a, \alpha, l) \in rels(p)\}$ 
10:  $r \stackrel{\pm}{\leftarrow} \{del_r((r, \alpha, l), p) \mid r = entry(inl), \exists (r, \alpha, l) \in rels(inl)\}$ 
11:  $r \stackrel{\pm}{\leftarrow} \{del_r((r, \alpha, l), p) \mid \exists r \in acts(inl), kind(r) = reply, \exists (r, \alpha, l) \in rels(inl)\}$ 
12: return  $r \stackrel{\pm}{\leftarrow} (del_a(a, p), del_a(entry(inl), p)) \cup \{del_a(r, p) \mid \exists r \in acts(inl), kind(r) = reply\}$ 

```

are mapped to the same preexisting one. At the implementation level, these issues are handled using the following techniques:

- Instead of deleting interface activities, we replace it by *nop* ones, and connect the predecessors (respectively successors) of a to this activity. It ensures that the preexisting relations are kept.
- Instead of replacing inlined variables by preexisting ones, we preserve their scope with the use of assignments. We copy the content of preexisting variables into the inlined ones at the beginning of the inlined behavior, and perform the opposite assignment at its end.

These techniques generate “ugly” processes, polluted with a lot of technical relations and activities. However, the inlined process are built for deployment purpose only, and then may not be readable for humans.

9.3.3 Intrinsic Limitations: Inlining is not a “Holy Grail”

In this section, we show how the INLINE algorithm is used to reduce the amount of exchanged data in the system. It does not impact the intrinsic complexity of a process: a quadratic process exchanging few data with the outside world will not take large benefits from inlining. As a consequence, the use of the INLINE algorithm can be seen as a final tune for a process, before deployment.

9.3.4 Integration in the ADORE Development Approach

We define the INLINE algorithm as an “optimization” step, which aims to enhance performances. As a consequence, we postpone its execution to the end of the step handling, as described in FIG. 9.5.

9.4 Using the INLINE algorithm on PICWEB

We consider here simplified process associated to the PICWEB case study, to fit the implementation used in the previously described benchmark. The *getPictures* process is represented in FIG. 9.6(a), and the associated *truncate* helpers in FIG. 9.6(b).

Before deploying its process, the user asks ADORE to perform the inlining. After executing the actions generated by the algorithm, an universe u' containing the inlined process is returned

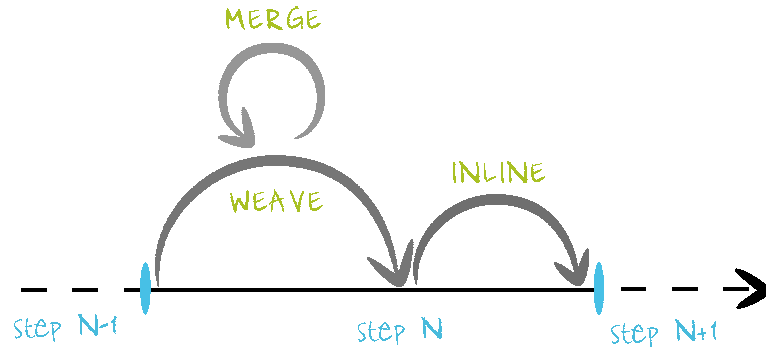
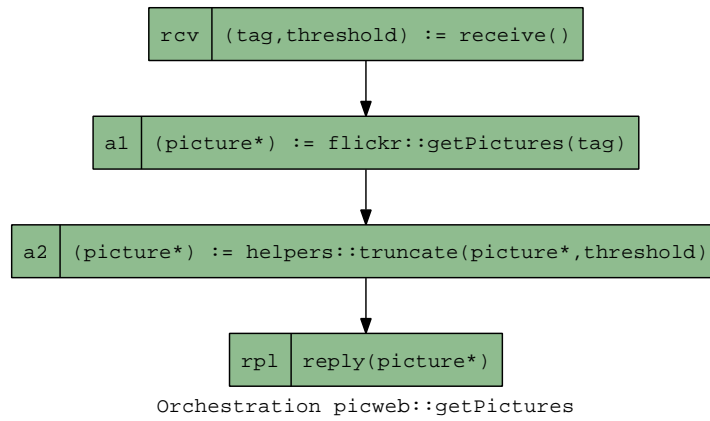


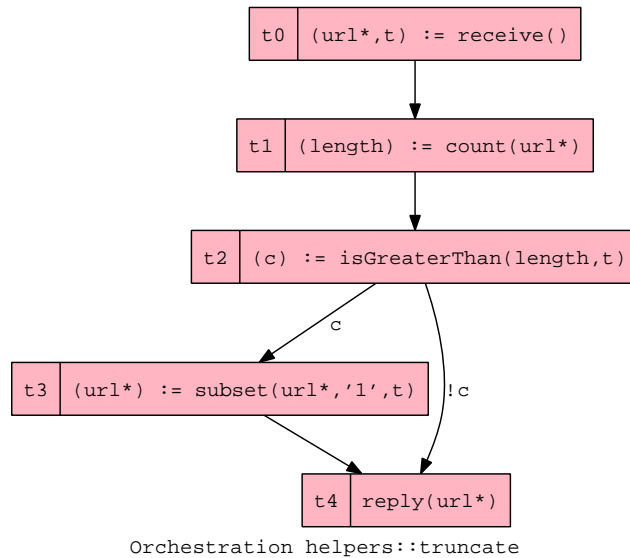
Figure 9.5: Integration of the INLINE algorithm in the ADORE approach

to the user. the obtained process is depicted in FIG. 9.7.

Let $u \in \mathcal{U}$, $getPictures \in procs(u)$, $truncate \in procs(u)$
 $S = \text{INLINE}(a_2, truncate, \equiv)$
 $u' = do^+(S, u)$

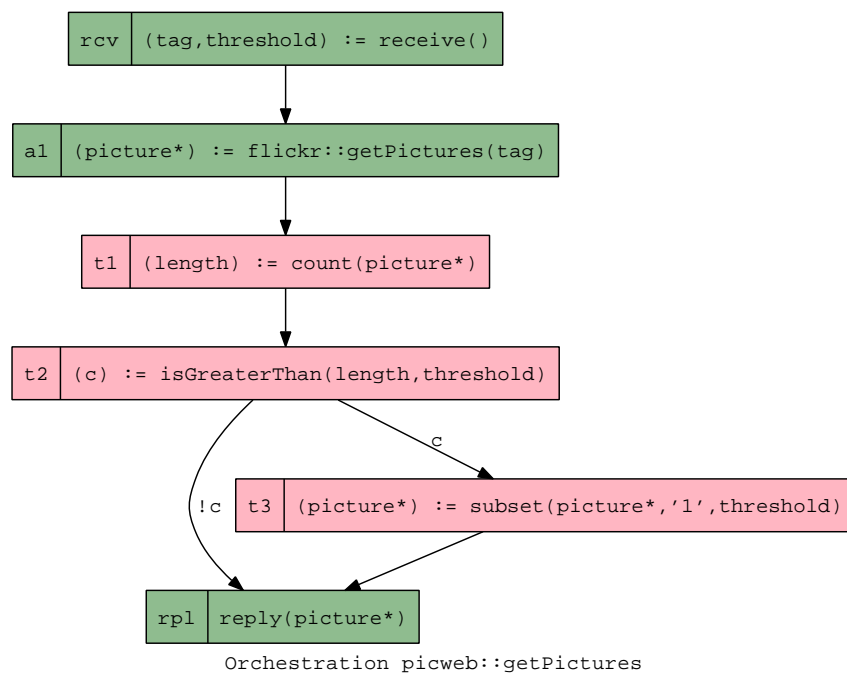


(a) Initial business process



(b) Helper process used to restrict the set cardinality

Figure 9.6: Simplified implementation of PICWEB (without the key registry)

Figure 9.7: Inlined version of the PICWEB *getPictures* process

Set-Driven Concerns: the DATAFLOW Algorithm

«Creativity comes from looking for the unexpected and stepping outside your own experience.»

Masaru Ibuka

Contents

10.1 Introduction	135
10.2 Motivations: Automatic Data-Driven Algorithms	136
10.2.1 Dataflow Refactoring	136
10.2.2 Array Programming	136
10.2.3 Grid-Computing: From Mashups to Scientific Workflows	136
10.3 Definition of the DATAFLOW algorithm	136
10.3.1 Running example: <i>Adder</i>	136
10.3.2 Principles	137
10.3.3 Description of the algorithm	139
10.4 Handling Real-Life Processes	142
10.4.1 Cosmetic Activities & Policy Make-Up	142
10.4.2 Towards Iteration Compositions	142
10.4.3 Integration in the ADORE Development Approach	143
10.5 Introducing Set-Concerns in PICWEB: $tag \mapsto tag^*$	143

10.1 Introduction

In the previous chapters, we described algorithms inspired by research on programming languages, software engineering and compilation. In this chapter, we go back to the fundamentals of SOA, and consider that business processes are not defined by computer scientists but by business experts. This idea is commonly used by several software design methodologies. For example, mashups-driven applications came from Web 2.0 movement and allow “*end-user programming of the Web*” [Wong and Hong, 2007], making web users able to compose data sources in a friendly way. Mashups invade scientific field of research like bioinformatic [Belleau et al., 2008]. At this level, dealing with dataset and the associated notions (e.g., iteration, loop termination, variant, invariant) is an error-prone and time consuming process. As mashups designers are not computer sciences specialists by essence, automatic tools dealing with iteration may ensure the correctness of builded flows. A preliminary version of this work is described in [Mosser et al., 2009a].

In this chapter, we define an iteration introduction algorithm. We propose to automatically enhance a process able to handle a scalar data d (more simple to define) into a process able to handle a dataset $d^* \equiv \{d_1, \dots, d_n\}$. The need for such algorithms is motivated in SEC. 10.2. The defined DATAFLOW algorithm is then described in SEC. 10.3. In SEC. 10.4, we describe

several leads useful to handle more complex data-driven concerns in the DATAFLOW introduction algorithm. Finally, we describe in SEC. 10.5 how this algorithm is used to automatically enhance the PICWEB running example.

10.2 Motivations: Automatic Data-Driven Algorithms

10.2.1 Dataflow Refactoring

Evolution techniques like code refactoring [Fowler, 1999] handle the redesign of legacy applications, following some well-known code rewriting techniques (interface extraction, inheritance, ...). These techniques are inspired by object-oriented concepts and typical object-oriented evolution use cases. Inside WSOA workflows, a typical evolution is to transform a data into a dataset and iterate computations over this dataset, *e.g.* for benchmarking purposes [Nemo et al., 2007]. To the best of our knowledge, no existing refactoring technique is able to automatically transform an orchestration working on a single data v into an orchestration able to handle a dataset $v^* \equiv \{v_1, \dots, v_n\}$.

- ↪ We propose to automate such a code refactoring mechanisms, in the context of ADORE processes. Experts can then focus on their business models, defined on scalar elements, and consequently more simple to express. An automated algorithm tackles the complexity of introducing an iteration where necessary to finally handle a dataset instead of a scalar.

10.2.2 Array Programming

Array programming [Hellerman, 1964] relies on efficient access and operation on arrays data. This paradigm defines several operators which work equally on arrays and on scalar data. Applications are then defined as the composition of operators. Current approaches try to introduce array programming concepts inside other existing paradigms like object-oriented languages for example [Mougin and Ducasse, 2003]. However, this very efficient paradigm is not immediately usable by non-specialists.

- ↪ Contrarily to this method, we propose to introduce the set concept at the model level to be able to reason on it, and then translate this concept into existing concepts (loop) in the targeted technology.

10.2.3 Grid-Computing: From Mashups to Scientific Workflows

In the particular domain of grid-computing, algorithms enacted on the grid are designed to handle very large datasets (*e.g.*, medical images, DNA sequences, weather models). Following the same paradigm than orchestrations, scientific workflows are supposed to be written by non-specialist users (*e.g.*, physician, neurologist). Grid workflows are data-intensive [Glatard, 2007] so there is a need for grid users to handle data-set composition at a high level of abstraction.

- ↪ We propose to automate the introduction of an iteration inside a process. The designer will express the process based on a scalar data, and the algorithm will automatically compute the associated iteration policy. With such a tool, non-specialist users will focus on their business (*e.g.*, detect multiple sclerosis symptoms in MRI) instead of the iteration technical considerations (*e.g.*, counter increment, nested structure, termination)

10.3 Definition of the DATAFLOW algorithm

10.3.1 Running example: *Adder*

As a running example, we define a simple *adder* orchestration illustrated in FIG. 10.1. This orchestration receives an invocation message containing two input string parameters x and y

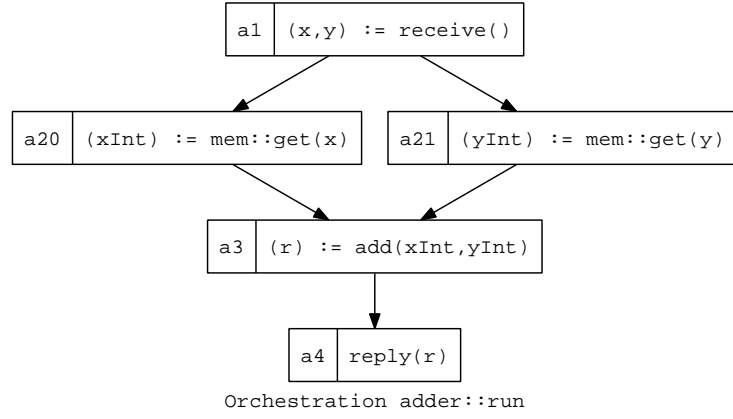


Figure 10.1: The *adder* orchestration: adds two integers stored in the memory

(a_1). It retrieves from a *memory* web service (a_{20}, a_{21}) the integer value associated to x in a variable x_{int} (resp. y_{int}) and returns (a_4) a result r (computed as $x_{int} + y_{int}$ in a_3).

Our goal is to produce through our DATAFLOW algorithm an orchestration $adder^{(x^*)}$ able to handle a set of values $x^* \equiv \{x_1, \dots, x_n\}$ instead of x . As the original process computes its result r from x_{int} and y_{int} , we want to compute a set of results $r^* \equiv \{r_1, \dots, r_n\}$, where $r_i = x_{int_i} + y_{int}$.

$$\begin{aligned}
 adder : \quad & String \times String \rightarrow Integer \\
 & (x, y) \mapsto x_{int} + y_{int} \\
 adder^{(x^*)} : \quad & String^* \times String \rightarrow Integer^* \\
 & (\{x_1, \dots\}, y) \mapsto \{x_{int_1} + y_{int}, \dots\}
 \end{aligned}$$

10.3.2 Principles

The principles used to define the DATAFLOW algorithm are simple. Based on an given variable v , it starts to identify v dataflow (as the transitive closure of x usages), and then builds an iteration policy based on such a dataflow. Following the W3C definition, an orchestration represents a “control-flow”. Identifying data flows inside an orchestration allows us to extract activities inside the control flow which derives from the usage of a given variable.

Dealing with Data Usages: *flow* & *core*. We extend the ADORE set of functions to deal with the transitive usage of a variable v . We divide this notion into two separated functions: (i) the *flow* function extracts the complete dataflow associated to v , and (ii) the *core* function restricts a flow to non-interface activities.

- *flow*: we define this function as a transitive closure of all activities *impacted* by the evolution of a variable v into a dataset v^* : $\forall a_i \in flow(p, v)$, a_i either uses v as input variable or a_i uses as input variable a variable transitively computed from v (computed with the *trans* function).

$$\begin{aligned}
 trans : \quad & \mathcal{P} \times \mathcal{V} \rightarrow \mathcal{A}^* \\
 & (p, v) \mapsto \{a_i \mid \exists \alpha \in acts(p), v \in inputs(\alpha), v' \in outputs(\alpha), a_i \in flow(p, v')\} \\
 flow : \quad & \mathcal{P} \times \mathcal{V} \rightarrow \mathcal{A}^* \\
 & (p, v) \mapsto \{a_i \mid v \in vars(a_i) \vee a_i \in trans(p, v)\}
 \end{aligned}$$

- *core*: we define the *core* of a dataset evolution as a set of activities built upon a *flow*, without interface activities of (i) output variables of the *firsts* activities of the *flow* (received from

the external world) and (ii) input variables of *lasts* activities of the *flow* (sent to the external world). It represents the set of activities $\{a_i, \dots, a_j\}$ defined in an orchestration o that needs to be included in an iteration policy to support the $v \mapsto v^*$ evolution.

$$\begin{aligned}
 inter? : \mathcal{A} \times \mathcal{V} &\rightarrow \mathbb{B} \\
 (a, v) &= (kind(a) \in \{receive, invoke\} \wedge v \in outputs(a)) \\
 &\quad \vee (kind(a) = reply \wedge v \in inputs(a)) \\
 core : \mathcal{P} \times \mathcal{A} &\rightarrow \mathcal{A}^* \\
 (p, v) &\mapsto flow(p, v) \setminus \{a_i \mid a_i \in firsts(flow(p, v)) \wedge inter?(a_i, v) \\
 &\quad \vee a_i \in lasts(flow(p, v)), \exists v' \in outputs(a_i) \wedge inter?(a_i, v')\}
 \end{aligned}$$

We use the *adder* example and the variable x to illustrate these two functions. Based on this example, the following activity sets are computed:

$$\begin{aligned}
 flow(adder, x) &= \{a_1, a_{20}, a_3, a_4\} \\
 core(adder, x) &= \{a_{20}, a_3\}
 \end{aligned}$$

A Four Step Algorithm. For a given orchestration o and a variable $v \in vars(o)$, the DATAFLOW mechanisms are the following: firstly, it extracts the evolution core $core(o, v)$, which represents the body of the computed loop; secondly a variable substitution $v \mapsto v^*$ is performed outside the *core* activities; thirdly the algorithm introduces special activities inside the *core* to interface the policy with the rest of the orchestration; and finally the order relation inside the orchestration is rearranged to take care of the policy introduction¹. FIG 10.3(b) illustrates the resulting workflow in the *adder* example case.

1. **Core Extraction.** Using previously defined functions and properties, the core associated to v in o is computed ($core(o, v)$). It represents the body of the computed loop we are going to build. The ADORE meta-model does not handle multiple iterations on the same activity. As a consequence, the core is analyzed and the algorithm rejects the evolution if at least one activity contained by the core is involved in another iteration. In our example, $core(adder, x) = \{a_{20}, a_3\}$. These two activities correspond to the body of the loop that processes the set of input data $\{x_1, \dots, x_n\}$.
2. **Variables Creation & Substitution.** In a generic way, the algorithm builds an iteration policy able to handle a dataset as *input*, and returns datasets as *output*. The dataflow may lead to the creation of several output sets (e.g., n different variables o_n transitively derived from the targeted variable v will lead to the creation of n output data sets o_n^*). The behavior defined in the *core* still works on the scalar variables, and the activities defined outside of this *core* will only see datasets (denoted as *input** and *output**). As a consequence, outside the *core*, these variables must be substituted (using substitution pairs) to match with the newly created datasets.

In the *adder* example, we have to perform two creations (x^* and r^*), and two substitutions: the input parameter x must be renamed as x^* , and the output variable r must be renamed as r^* . The associated substitutions $\theta = \{x \leftarrow x^*, r \leftarrow r^*\}$ must be applied on activities defined outside of the core (message reception as a_1 and response sending as a_4).

3. **Policy Creation & Adaptation** A **serial** iteration policy is created, containing the *core* activities. The data-link between the *input** content and the preexisting *input* variable is handled by the policy semantic (defined for all $input \in input^*$). Inside the policy, a special assignment activity (named “*sweller*”) is defined to store the scalar output currently computed in *output**. The order relation is enriched to push the *sweller* after its associated last activity (e.g., the *sweller* associated to the output dataset out_i^* is defined as a follower of the last activity which uses out_i as output). If there is no output variable produced by the

¹According to the meta-model constraints, a block of activities used inside a loop must be well-formed PROP. 4.21.

core, there is no need for a *sweller* (there is nothing to store!). As a consequence, this step is simply skipped in such a situation.

In the *adder* example, the *sweller* is defined as an assignment from r into r^* using the *append* function (s_1). The created policy contains the activity set $\{a_{20}, a_3, s_1\}$. A new order relation $a_3 \prec s_1$ must be added to push the sweller associated to x^* after the last activity of the dataflow which handles x (i.e., a_3).

4. **External Activities Reordering.** The order relation $rels(o)$ defined inside o may be inconsistent for now. The ADORE meta-model constraints ensure that the content of a policy is well-formed. Activities and relation leading to ill-formed blocks must be reordered to satisfy this constraint. We illustrate the activity reordering in FIG. 10.2. Let *ext* an activity which is not part of the policy, and *body* an internal activity of the policy. We identified three erroneous situations: (i) a relation between *ext* and *body*, (ii) a relation between *body* and *ext*, and finally (iii) the two previous situations simultaneously. A relation $ext < body$ is pulled before the beginning of the policy (see $a_1 \prec a_2$ in FIG. 10.2(b)). The reciprocal situation ($body < ext$) pushes the activity after the end of the policy (see $a_6 \prec a_5$ in FIG. 10.2(b)). Finally, when both situations are encountered on the same activity a , the policy will simply absorb a (see a_4 in FIG. 10.2(b)). These mechanisms are intrusive and break the preexisting execution semantic. As a consequence, all these decisions are returned to the designer, for information purpose.

The *adder* example obtained before reordering (FIG. 10.3(a)) contains two erroneous relations: $a_{21} \prec a_3$ and $a_3 \prec a_4$. According to these two relations, the defined iteration policy contains an ill-formed block, and then is not valid according to the meta-model constraint. The reordering step pulls a_{21} before the beginning of the iteration and on the other side pushes a_4 at its end. The resulting orchestration $adder(x^*)$ is described in Fig 10.3.

10.3.3 Description of the algorithm

We describe in ALG. 7 the implementation of the DATAFLOW algorithm principles. The algorithm takes as input a process p and a variable v . It starts by initializing a context (line 2), used to generate the symbols associated to the potential “swellers” activities. We denote as C the *core* of the iteration policy, and as *outs* the variables representing outputs of the policy. In lines 4 and 5, we generate the new datasets variables associated to v and *outs*, using a star postfix convention. Then, the “swellers” activities are created (a “sweller” is generated from its associated *outs* variable) and correctly inserted at the end of the policy. The future content of the policy (C and the “swellers”) are stored (line 8). Finally, the activity reordering step is performed in lines 13 → 15. The iteration policy (containing the previously identified activity and the absorbed ones) is finally created (line 16), and the action list is returned to the caller.

Limitations. The previously described algorithm suffers from two limitations: (i) a fixed data structure and (ii) no reasoning over the available preexisting relations.

- **Fixed Data Structure.** The data model associated to ADORE is simplistic. Considering that we do not reify the *type* associated to a variable, we use a boolean flag to mark a variable as a data set. Consequently, it is not possible to represent directly a “set of set” in ADORE. Let i^* a set of integers, modeled in adore as a tuple $(i^*, integer, true)$. To define I as a set of sets of integers, one can use the following tuple $(I, arrayOfInteger, true)$, assuming that the *arrayOfInteger* type is binded to such a data structure. This limitation is a part of the perspective about data structure exposed in SEC. 13.4.
- **No Relations Reasoning.** The reordering mechanisms are technical and do not reason about the relations they are reordering. For example, *onFailure* or *guard* relations may require a particular attention when reordered. As a consequence, this algorithm only sketches the automatic introduction of data set iterations in a process. However, its implementation was sufficient to deal with very large and complex business processes defined in the CCCMS case study (CHAP. 11).

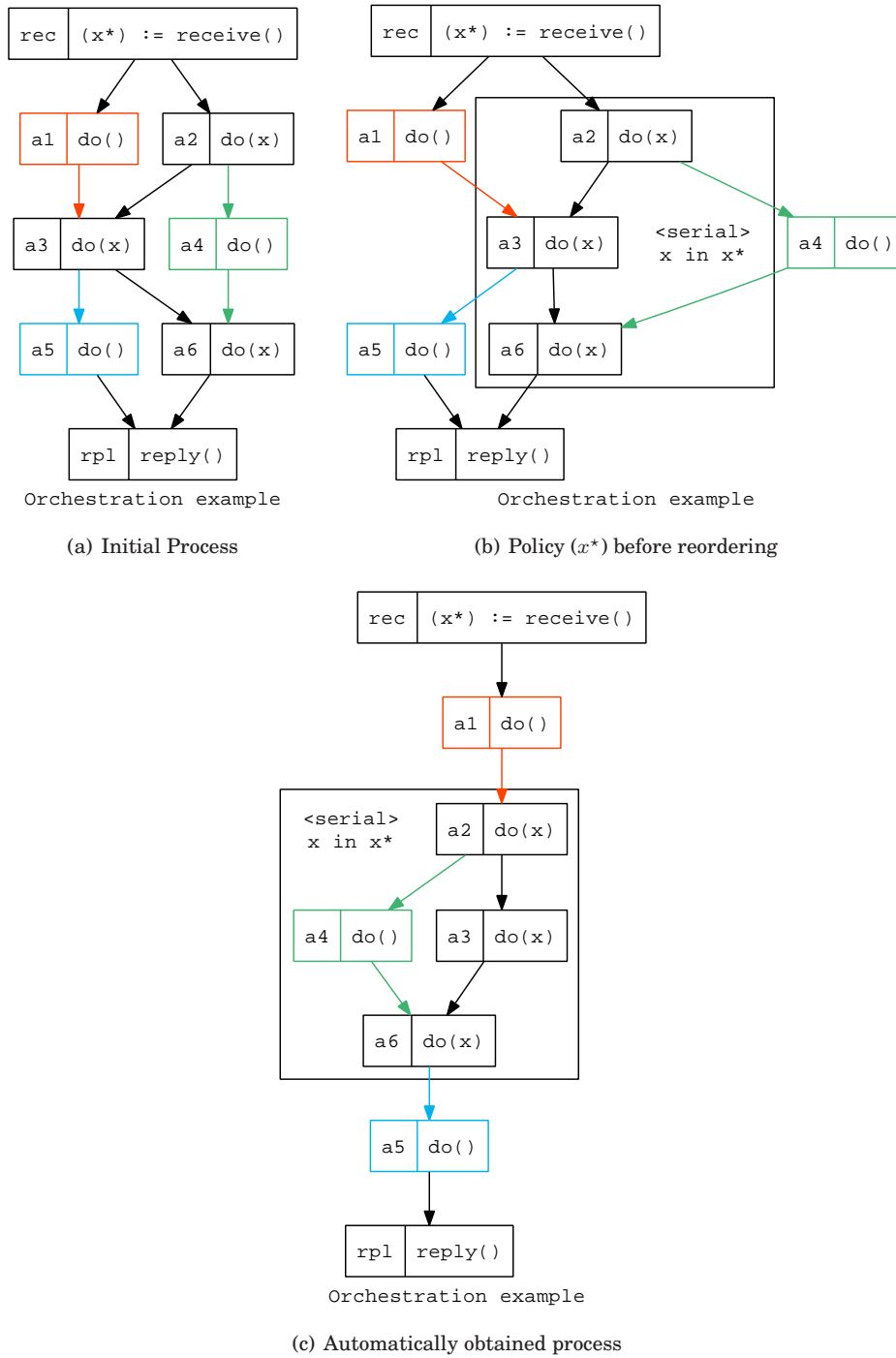
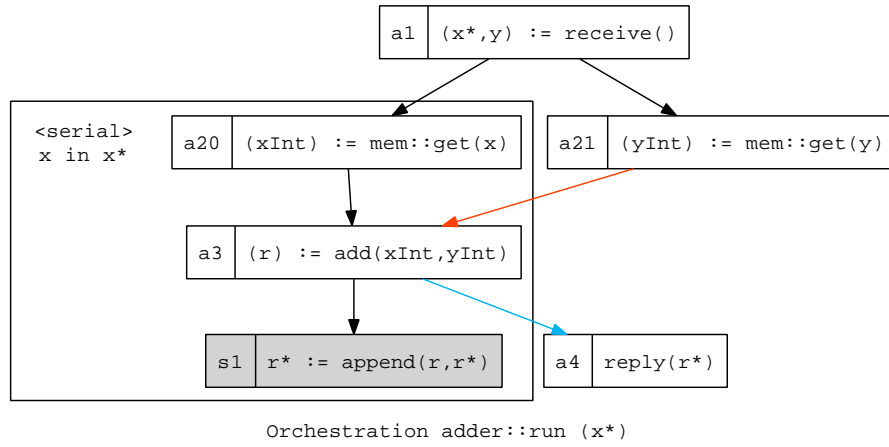
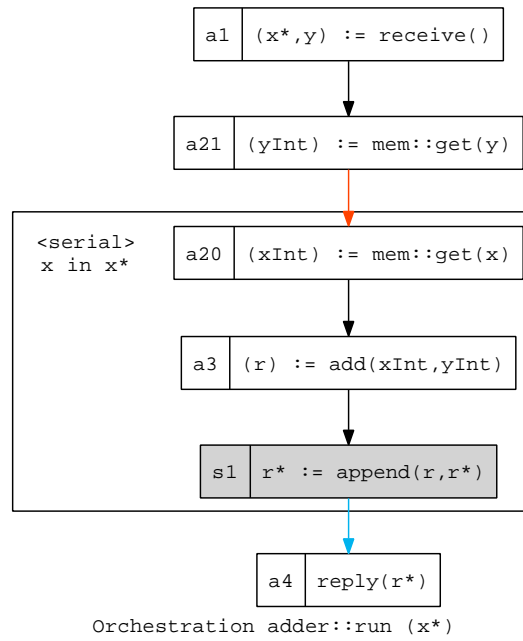


Figure 10.2: Illustrating the DATAFLOW reordering mechanisms

(a) Raw version of $adder^{(x^*)}$ (before activity re-ordering)

(b) Final version (after automatic re-ordering)

Figure 10.3: Automatically built $adder^{(x^*)}$ process: “*adder* for a set of x ”.

Algorithm 7 DATAFLOW: $p \times v \mapsto \text{actions}$ **Require:** $p \in \mathcal{P}, v \in \text{vars}(p)$ **Ensure:** $r \in \text{Actions}^*_<$

```

1:                                     { (Step 1) Initialization }
2:  $ctx \leftarrow \text{context}(), C \leftarrow \text{core}(p, v), \text{outs} \leftarrow \{var \mid \exists \alpha \in \text{lasts}(C), var \in \text{outputs}(\alpha)\}$ 
3:                                     { (Step 2) Variable Creation & Substitutions }
4:  $\theta \leftarrow \{name(var) \leftarrow name(var)^* \mid \exists var \in \text{outs} \cup \{v\}\}$ 
5:  $r \leftarrow \{add_v((name(var)^*, type(var), true), p) \mid \exists var \in \text{outs} \cup \{v\}, r \vdash \{r(\alpha, \alpha\theta, p) \mid \exists \alpha \in \text{acts}(p) \setminus C\}$ 
6:                                     { (Step 3) Policy Content Creation & Adaptation }
7:  $r \vdash \{add_a((gensym(var, ctx), assign(append), \{n, n^*\}, \{n^*\}), p) \mid var \in \text{outs}, n = name(var)\}$ 
8:  $r \vdash \{add_r(\alpha \prec gensym(var, ctx), p) \mid \alpha \in \text{lasts}(C), var \in \text{outputs}(\alpha)\}$ 
9:  $r \vdash \{add_r(gensym(var, ctx) \prec next, p) \mid \alpha \in \text{lasts}(C), var \in \text{outputs}(\alpha), \alpha < next \in \text{rels}(p)\}$ 
10:  $PolActs \leftarrow C \cup \{gensym(var, ctx) \mid var \in \text{out}\}$ 
11:                                     { (Step 4) Reordering }
12:  $Body \leftarrow \text{internals}(C), Ext \leftarrow \text{acts}(p) \setminus Body$ 
13:  $r \vdash \{add_r((e, f, tag), p) \mid e \in Ext, \exists f \in \text{firsts}(C), \beta \in Body, \exists (e, \beta, tag) \in \text{rels}(p), \nexists \beta' \in Body, \beta' \rightarrow e\}$ 
14:  $r \vdash \{add_r((l, e, tag), p) \mid e \in Ext, \exists f \in \text{firsts}(C), \beta \in Body, \exists (\beta, e, tag) \in \text{rels}(p), \nexists \beta' \in Body, e \rightarrow \beta'\}$ 
15:  $Absorbed \leftarrow \{e \mid \exists \beta \in Body, \exists \beta' \in Body, \exists e \in Ext, \beta \rightarrow e \wedge e \rightarrow \beta'\}$ 
16:  $r \vdash (add_i(name(v)^*, name(v), serial, PolActs \cup Absorbed), p)$ 
17: return  $r$ 

```

10.4 Handling Real-Life Processes

10.4.1 Cosmetic Activities & Policy Make-Up

Cosmetics Activities. The algorithm defined in the previous section is a generic algorithm which automatically formats the output of the iteration policy as a list corresponding to a concatenation of the results dataset. In some cases, this generation is too basic and it does not reflect the expected semantics of the dataset concern. For instance when working on numerical data, a common need is to compute a global statistical result such as the average of a set of results rather than handling individual values. ADORE interference detection rules set can detect inconsistencies introduced by the make-up step (e.g. parallel write access), but it is under user's responsibility to solve those conflicts based on the application semantic.

To handle this diversity of possible algorithm specialization, we consider a set of “cosmetic” activities, like flattening a list or computing an average. The user can manually insert a cosmetic activity associated to an iteration policy to perform final optimization. As a consequence, the automatically computed behavior is a *default* one, which can be customized to fit user needs. For example, if the contents of the iterations allows it, one can decide to change the kind of iteration from **serial** to **parallel**.

10.4.2 Towards Iteration Compositions

An unresolved issue of the DATAFLOW algorithm is the handling of multiple datasets. For example, one may decide to enhance the *adder* process for both x and y input variables. Such a feature is not supported by ADORE. The grid computing community uses dataflow language (e.g., SCUFL [Oinn et al., 2004], GWENDIAN [Montagnat et al., 2009]) designed to handle data sets. Based on these descriptions, dataflow are analyzed and their enactment on a grid can be optimized [Glatard, 2007]. Data composition operators were defined in [Montagnat et al., 2006] allowing composition through iteration strategy (one to one, one to many). Using these operators, the designer can decide to model the following compositions:

$$\begin{aligned}
x^* \odot y^* &\rightsquigarrow \{x_1 + y_1, \dots, x_n + y_n\} \\
x^* \otimes y^* &\rightsquigarrow \{x_1 + y_1, \dots, x_1 + y_m, x_2 + y_1, \dots, x_n + y_m\}
\end{aligned}$$

These operators (\otimes , \odot) can be naturally composed. We expose as a perspective the usage of iteration operator composition to address this issue in ADORE.

10.4.3 Integration in the ADORE Development Approach

The DATAFLOW algorithm is, like the INLINE one, a finalization step. To be semantically correct, the computed *flow* must take care of the activities introduced by the WEAVE directives. Consequently, it must be executed after. Considering that the INLINE algorithm presented in this document does not handle iteration policies, we postpone the execution of the DATAFLOW computation at the end of the development step. The final version of the approach is described in FIG. 10.4.

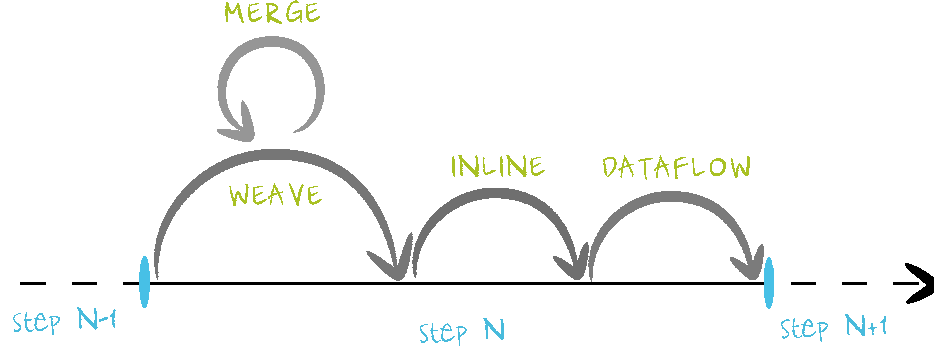


Figure 10.4: Integration of the DATAFLOW algorithm in the ADORE approach

10.5 Introducing Set-Concerns in PICWEB: $tag \mapsto tag^*$

In this section, we consider the PICWEB process obtained at the end of CHAP. 8. This process, depicted in FIG. 8.7 corresponds to the realization of the complete requirements expressed in the example description. This process is also the model associated to the existing implementation of folkwonomy exploration in the legacy JSEDUITE application (CHAP. 12).

We decide to enhance the process to deal with a set of tags, instead of a scalar one. To lighten the figures, we use a shortened graphical notation for each activity (represented by a single box). Based on the following computation associated to $core(getPictures, tag)$, the algorithm build (without reordering) the process depicted in FIG. 10.5.

$$core(getPictures, tag) = \{a_2, a_3, aP_1, aP_2, c_1, c_2, c_3, i, t_2, t'_2\}$$

The reordering step *absorbs* the activities defined *in-the-middle* (i.e., $\{iO_1, iI_1, t, t_3, t'_3\}$). The final result is depicted in FIG. 10.6. Consequently, we automatically obtain a process able to handle a dataset instead of a scalar data:

$$\begin{aligned} \text{Let } u &\in \mathcal{U}, \text{ getPictures} \in \text{procs}(u) \\ \leftarrow S &= \text{DATAFLOW}(\text{getPictures}, tag) \\ u' &= do^+(S, u) \end{aligned}$$

$$\begin{aligned} \text{getPictures} \in \text{procs}(u) : \quad &String \times Integer \rightarrow ArrayOfPictures \\ &(tag, threshold) \mapsto pictures \end{aligned}$$

$$\begin{aligned} \text{getPictures} \in \text{procs}(u') : \quad &String^* \times Integer \rightarrow ArrayOfPictures^* \\ &(tag^*, threshold) \mapsto pictures^* \end{aligned}$$

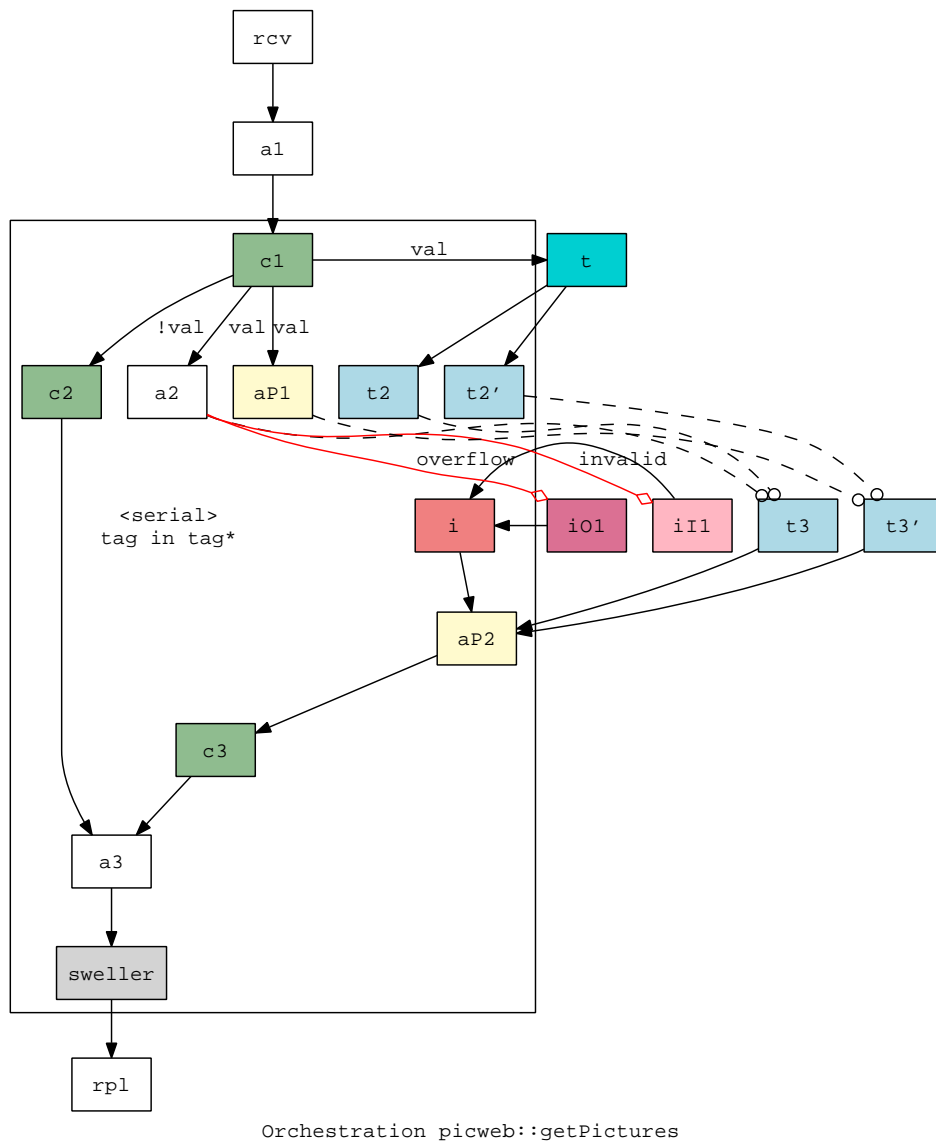
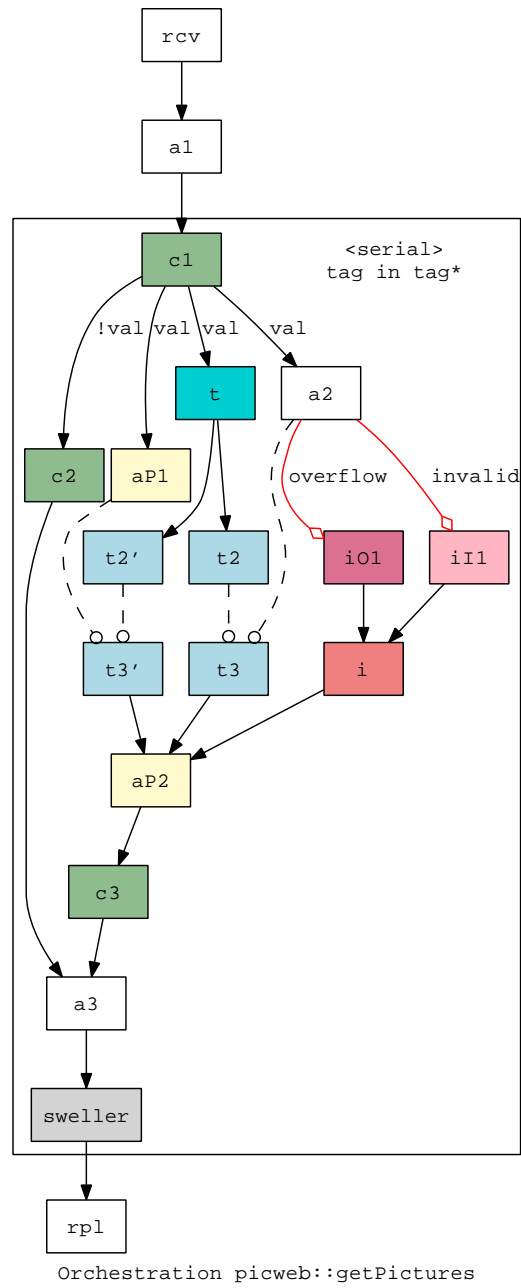
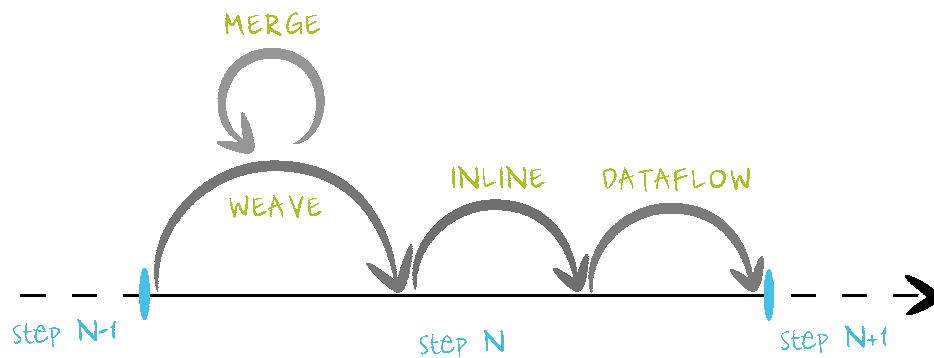


Figure 10.5: Illustrating the reordering mechanisms need on the PICWEB example

Figure 10.6: Automatically obtained PICWEB process ($tag \mapsto tag^*$)

Summary

In this part, we defined four different algorithms used to perform “compositions” in the context of business processes. Moreover, we scheduled these algorithms into a computer-aided development approach.



The WEAVE algorithm is the first step of this approach. It aims to support the integration of *fragments* into other processes. This algorithm works on a set of parallel directives, and generates the actions to execute to perform the expected integrations. Moreover, relying on the WEAVE algorithm to enhance business processes ensures the respect of several composition properties, such as determinism, path and conditions preservation, Interference detections mechanisms are used to detect issue in the composed result.

The MERGE algorithm handles the well-known Shared Join Points problems, in the context of business processes. It is defined to support designers when several fragments share the same target location. The algorithm allows the composition of these fragments in an oblivious way (*i.e.*, without any considerations about the composed system), and lets designers focus on a restricted scope (the shared fragments only, instead of the final system) to assess the computed behavior.

The INLINE algorithm is inspired by the eponymous compilation technique. For performance reasons, one can decide to *inline* a process inside another one, *i.e.*, replace its invocation activity by the actual code. We use an implemented benchmark to verify this assumption at the implementation level, and surprisingly, the actual results are better than the theoretical ones.

Finally, the DATAFLOW algorithm is used to support non-specialists while they handle datasets. Using these mechanisms, a business expert can design a process working on a scalar data d , and ask the algorithm to automatically refactor the process into a process able to handle a dataset $d^* = \{d_1, \dots, d_n\}$.

~> A well known scientific adage states: “*Theory guides. Experiment decides.*”. In the next part, we use the defined algorithms and the associated development approach to handle two real-life applications.

Part IV

Applications

Notice: This part describes the usage of ADORE to handle two different real-life applications. The first one (CHAP. 11) is designed from scratch, at the model level. The second one (CHAP. 12) is a daily used application, re-engineered with ADORE

A Car Crash Crisis Management System (CCCMS)

«Another feature that everybody notices about the universe is that it's complex.»

Seth Lloyd

Contents

11.1 Introduction	151
11.2 Realizing the CCCMS	152
11.2.1 Requirements	152
11.2.2 Modeling Use Cases as Orchestrations	152
11.2.3 Modeling Extensions as Fragments	154
11.3 Zoom on Non-Functional Concerns	155
11.3.1 Modeling Non-Functional Concerns as Fragments	155
11.3.2 Discovering Fragments Target using Logical Predicates	155
11.4 Taming Orchestration Design complexity	156
11.4.1 Modeling the Complete CCCMS	156
11.4.2 MERGE to Assess Shared Join Points	157
11.4.3 WEAVE to Support Complex Behavior Definition	158
11.4.4 Interference Detection Mechanisms in Action	158
11.5 Quantitative Analysis	161
11.5.1 Coarse-Grained Structural Complexity	161
11.5.2 Entity Provenance	162
11.5.3 Cognitive Load	162
11.6 Conclusions	163
11.6.1 Approach Intrinsic Limitations	163
11.6.2 Summary	164

11.1 Introduction

This chapter describes a model-driven case study realized with ADORE for a TAOSD special issue on Aspect Oriented Modeling. This special issue was based on a common case study, a Crisis Management System (CMS). The goal of this special issue is to compare existing *Aspect Oriented Modeling* approaches between each other, based on the same example. In [Kienzle et al., 2009c], authors describe the requirements of their CMS. According to the definition given by this case study, a CMS is “a system that facilitates coordination of activities and information flow between all stakeholders and parties that need to work together to handle a crisis”. Many types of crisis can be handled by such systems, including terrorist attacks, epidemics, accidents. To illustrate the case study, they provide an instance of a CMS in the context of car accidents. We decide to model

this particular instance using ADORE, instead of inventing a new one. Obviously, this chapter cannot exhaustively describe the implementation of such a complex system. A more detailed description is available in the associated publication [Mosser et al., 2010b] and on the associated website. The requirements define this system as the following:

“The Car Crash CMS (CCCMS) includes all the functionalities of general crisis management systems, and some additional features specific to car crashes such as facilitating the rescuing of victims at the crisis scene and the use of tow trucks to remove damaged vehicles.” [Kienzle et al., 2009c]

In this chapter, we describe in SEC. 11.2 how the requirements are modeled in ADORE. We describe the realization of non-functional concerns in SEC. 11.3, and SEC. 11.4 illustrates how ADORE tames the complexity of complex business processes design following a separation of concerns approach. We provide in SEC. 11.5 a quantitative analysis of the CCCMS, based on process metrics comparisons. Finally, SEC. 11.6 concludes this chapter by summarizing strengths and weaknesses of ADORE in front of this case study.

11.2 Realizing the CCCMS

11.2.1 Requirements

Considering ADORE as a behavior-driven AOM approach, we focus on the behaviors described in the requirements. The document defines ten use cases, described using textual scenario. Each scenario defines first a *main success scenario* which represents the normal flow of actions to handle a crisis (e.g., retrieve witness identity, contact firemen located near to the crash location). Then, a set of *extensions* are described to bypass the normal flow when specific actions occur (e.g., witness provides fake identification, firemen are not available for a quick intervention).

Based on these requirements, we decide (i) to model main success scenarios as orchestrations of service, (ii) to model business extensions and non-functional concerns as fragments, and finally (iii) use composition algorithm to tame the complexity of designing the complete CCCMS.

11.2.2 Modeling Use Cases as Orchestrations

The use cases defined in the requirements document are treated as informal specifications of service orchestrations. We use ADORE to model orchestrations. It is important to note that the focus is not on modeling the internal behavior of services or activities, but on the modeling of orchestrations. In the approach, a use case is realized as an orchestration of services. The main success scenario in a use case is realized as a base orchestration.

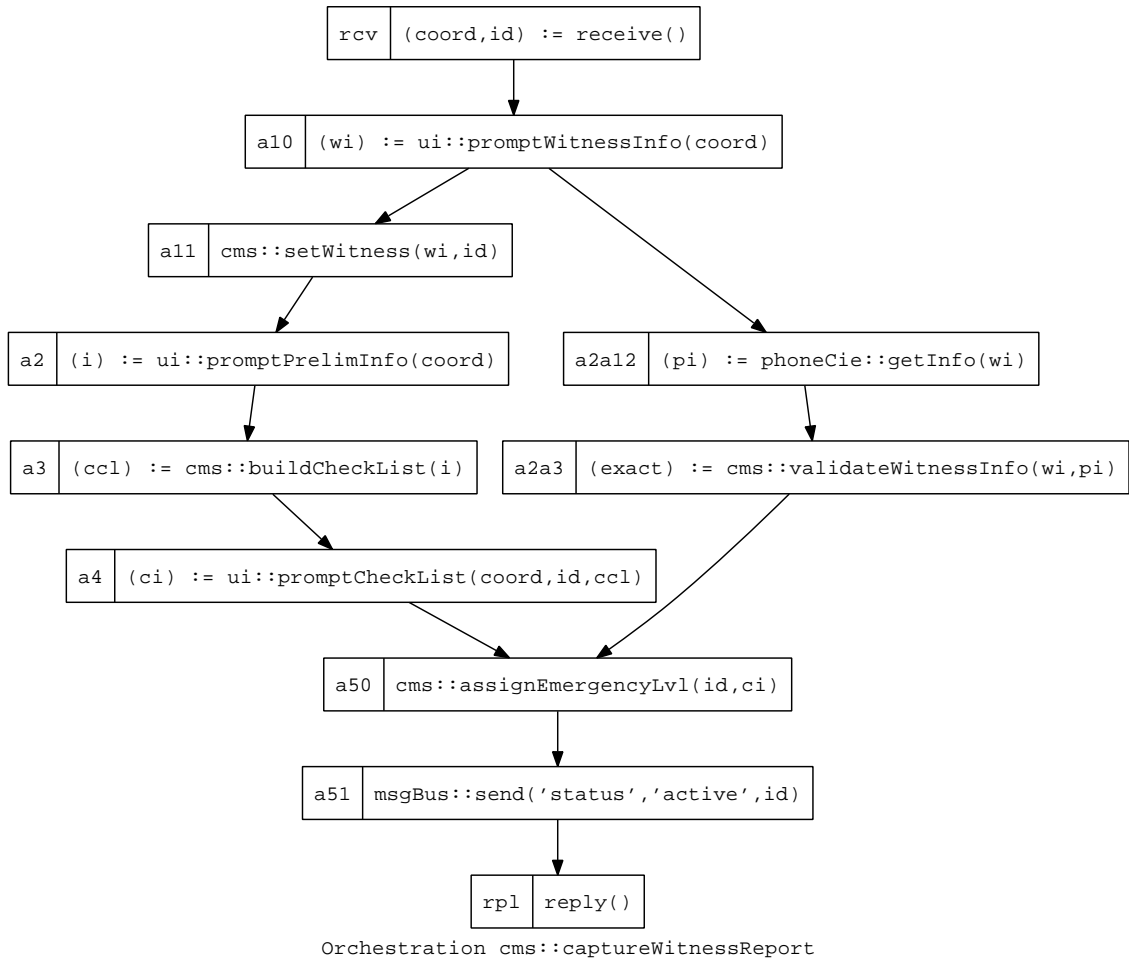
We illustrate this choice using the “*Capture Witness Report*” use case (#2). Below is the description of the main success scenario for this use case, extracted from the requirement document:

“Coordinator requests Witness to provide his identification.

- 1. Coordinator provides witness information to System as reported by the witness.*
- 2. Coordinator informs System of location and type of crisis as reported by the witness.*
In parallel to steps 2 – 4:
 - 2a.1 System contacts PhoneCompany to verify witness information.*
 - 2a.2 PhoneCompany sends address / phone information to System.*
 - 2a.3 System validates information received from the PhoneCompany.*
- 3. System provides Coordinator with a crisis-focused checklist.*
- 4. Coordinator provides crisis information to System as reported by the witness.*
- 5. System assigns an initial emergency level to the crisis and sets the crisis status to active.*

Use case ends in success.” [Kienzle et al., 2009c]

The above scenario is realized as an orchestration of four service providers: (i) **cms** represents the services provided by the *System* actor, (ii) **phoneCie** represents the services provided by the *PhoneCompany* actor, (iii) **ui** represents services used to interact with the *Coordinator* actor and (iv) **msgBus** represents services used to broadcast the mission status.



Step	Acts.	Step	Acts.	Step	Acts.	Step	Acts.
1	{a ₁₀ , a ₁₁ }	2	a ₂	3	a ₃	4	a ₄
5	{a ₅₀ , a ₅₁ }	2a.1	a _{2a12}	2a.2	a _{2a12}	2a.3	a _{2a3}

Step ↔ Activities correspondences

Figure 11.1: Orchestration representation for the *Capture Witness Report* use case (#2)

Fig. 11.1 shows a graphical ADORE model of the realization. The first step in the use case scenario is realized by two activities: (i) the **promptWitnessInfo** operation provided by the **ui** service is invoked (a₁₀, this operation requests and records witness information) and (ii) the **setWitness** operation provided by the **cms** is called (a₁₁) to add the witness information to the current crisis. Steps 2, 3 and 4 in the main use case scenario (getting crisis preliminary information, building a crisis-dedicated checklist and prompting for check list answers) are each realized by a single activity: a₂, a₃, a₄ respectively. Steps 2a.1, 2a.2 and 2a.3 must be done in parallel with steps 2 – 4. Steps 2a.1, and 2a.2 are realized by the activity a_{2a12}, while step 2a.3 is realized by a_{2a3}. Finally, step 5 is realized by two activities in the ADORE model: a₅₀, which assigns the emergency level, and a₅₁, which sends a message on the message bus indicating that the status of this crisis is now “active”.

11.2.3 Modeling Extensions as Fragments

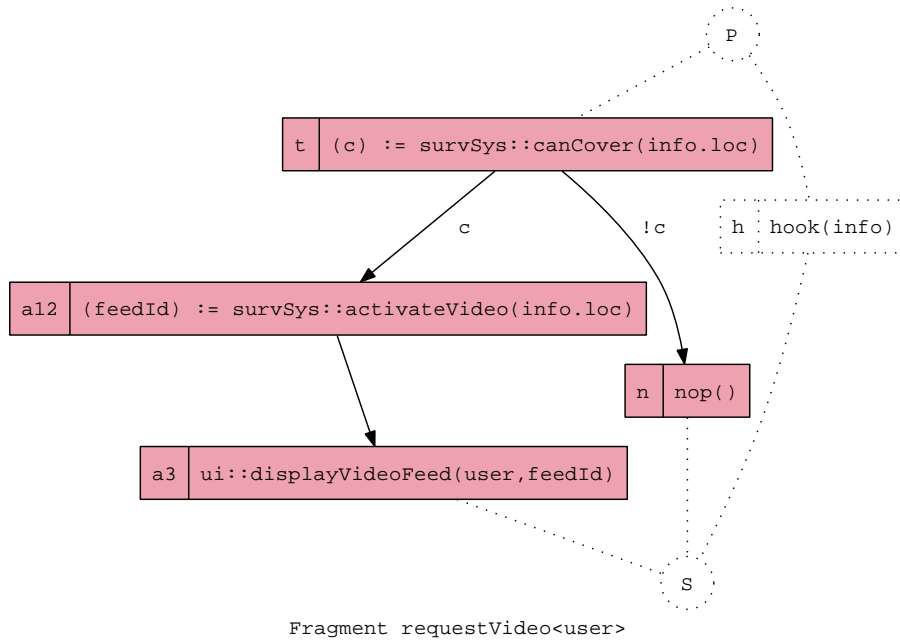
The *Capture Witness Report* use case describes six extensions to the main success scenario. These extensions deal with abnormal situations, such as the disconnection of the witness phone call or the detection of fake information given by the witness. We take as an example the extension #3a, which describes how the main success scenario is enhanced if a camera-surveillance system is available on the crisis location. This extensions is stated as follows:

“In parallel to steps 3 – 4, if the crisis location is covered by camera surveillance:

3a.1 System requests video feed from SurveillanceSystem.

3a.2 SurveillanceSystem starts sending video feed to System.

3a.3 System starts displaying video feed for Coordinator.” [Kienzle et al., 2009c]



Step	Acts.	Step	Acts.	Step	Acts.
3a.1	a_{12}	3a.2	a_{12}	3a.3	a_3

Step \leftrightarrow Activities correspondences

Figure 11.2: *RequestVideo* Fragment dealing with *Capture Witness Report* extension #3a

Figure 11.2 shows the fragment that realizes the above extension. The right branch of the fragment contains the hook, h , that represents the behavior in the targeted orchestration to which the fragment will be attached. The left branch of the fragment represents the additional behavioral described in the use case the extension #3a. In parallel to the behavior described by the hook (h) the system first determines if the surveillance system can cover the crisis area (t). If this functionality is available for this location, the process requests a video feed (steps 1 and 2 in the use case extension are aggregated in a single ADORE activity a_{12}) and then broadcasts it to the *Coordinator* interface (a_3)¹. This fragment is woven on the block of activities $\{a_3, a_4\}$, considering that the associated requirements extension target the steps 3 and 4 of the main success scenario.

¹One may notice that the *user* variable used in the a_3 activity is undefined in the fragment. This variable is *replaced* after weaving by the *coord* one which pre-exists in the *captureWitnessRepost* process. By using such a mechanism, one can use in a fragment a variable which is not *hooked*. Assuming that this usage is *read-only*, it does not jeopardize the compositional properties ensured by the WEAVE algorithm.

11.3 Zoom on Non-Functional Concerns

11.3.1 Modeling Non-Functional Concerns as Fragments

ADORE can be used to model non-functional (NF) properties that can be realized as system behaviors. We realized three NF-concerns in the context of the CCCMS case study: (i) persistence, (ii) security and (iii) statistical logging. We define five fragments to implement these concerns in the CCCMS.

- *Status Persistence.* The *persistence* concern involved two fragments: *logCreate* (transforming a transient entity into a persistent one) and *logUpdate* (logging status information for a persistent entity). We make the choice to handle employees (both **cmsEmployee** and **worker** entities) as persistent resource. As a consequence, the *logCreate* (resp. *logUpdate*) fragment must be woven each time a service invocation creates (resp. uses) such an entity.
- *Security.* We realized a part of the *security* concerns through the following scenario: “an idle employee must be re-authenticated”. We define a dedicated fragment (*authenticateWhenIdle*) to deal with this concern, and weave it on all activities interacting with an employee through the user-interface.
- *Statistical Logging.* We realized this property by storing the execution time of a given action. We differentiate *normal* execution (the action succeeds, *logTime*) and *exceptional* execution (the action failed, *logError*).

We focus now on the *statistical logging* property description, as defined in the requirements. The two fragments realizing this property are depicted in FIG. 11.3.

- “The system shall record the following statistical information on both on-going and resolved crises: rate of progression; average response time of rescue teams; individual response time of each rescue team; success rate of each rescue team; rate of casualties; success rate of missions.”
- “The system shall provide statistical analysis tools to analyze individual crisis data and data on multiple crises.” [Kienzle et al., 2009c]

Functional vs Non-Functional: a Fuzzy Border. The requirement document defines at the business scenario level a non-functional property! In use cases #1 (“Resolve Crisis”) and #3 (“Assign Internal Resource”), equivalent scenario extensions dealing with user authentication are defined (#1.1a and #3.1a). ADORE does not make any differences between functional and non-functional concerns, focusing on behavioral composition. As a consequence, we naturally reified this concern as a dedicated fragment (*mustAuthenticate*), shared with the two associated business processes.

11.3.2 Discovering Fragments Target using Logical Predicates

ADORE does not provide any surface mechanisms to deal with pointcuts expressiveness. But the underlying logical foundations of ADORE allow one to query the model using logical predicates. It obviously implies that the designer knows the first-order logic paradigm. But the immediate benefits are the unbounded possibilities of pointcut description [Rho et al., 2006]. We concretely illustrate this situation using the NF-properties example. The associated fragments need to be woven on many different activities, spread over the CCCMS models.

The *persistence* concern impacts the variables representing employees of the CCCMS company. The *logCreate* fragment is used to transform a transient entity into a persistent one. As a consequence, we need to identify all the activities in the CCCMS which creates an employee (i.e., using as output a variable of type **cmsEmployee** or **worker**). The reciprocal fragment *logUpdate*

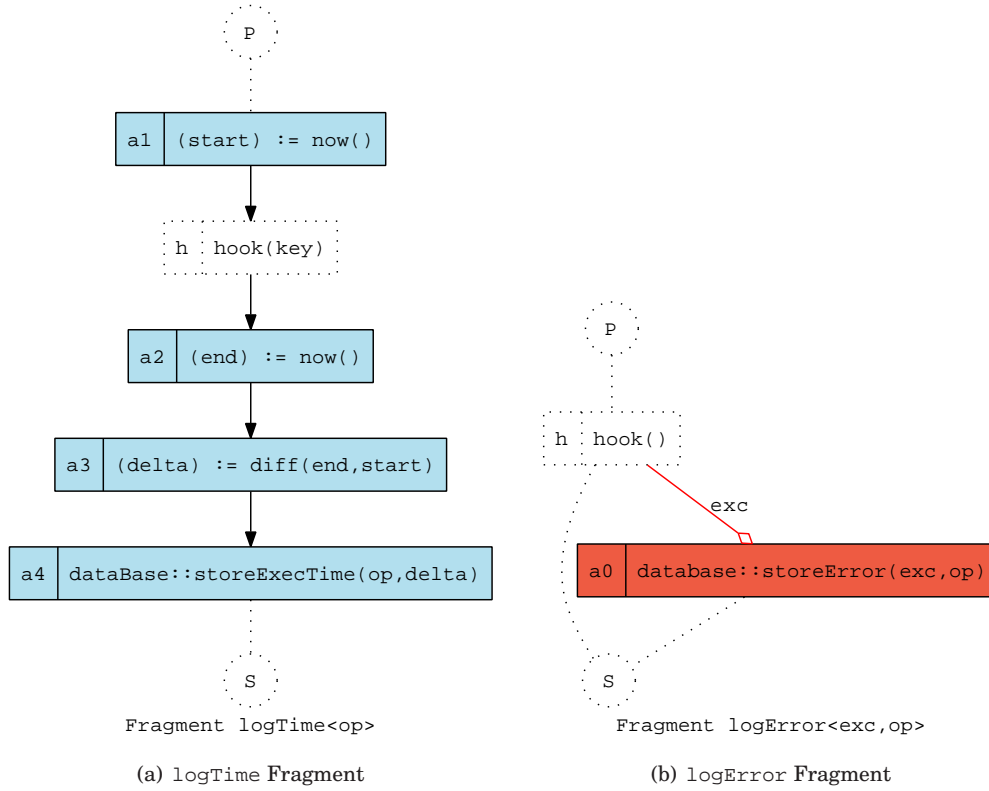


Figure 11.3: Non-functional fragments storing response times and errors

changes the status of a persistent entity, and must be woven on activities using employee as input. Such activities can be easily identified as the following:

$$\begin{aligned}
 \logCreate &\rightsquigarrow \{\alpha \mid \alpha \in \mathcal{A}, \exists v \in \text{outputs}(\alpha), \text{type}(v) \in \{\text{cmsEmployee}, \text{worker}\}\} \\
 \logUpdate &\rightsquigarrow \{\alpha \mid \alpha \in \mathcal{A}, \exists v \in \text{inputs}(\alpha), \text{type}(v) \in \{\text{cmsEmployee}, \text{worker}\}\}
 \end{aligned}$$

11.4 Taming Orchestration Design complexity

11.4.1 Modeling the Complete CCCMS

To realize the CCCMS as a set of business processes, we design 12 orchestrations of services, and 24 fragments. These artifacts represent 276 activities ordered by 268 relations. To express the composition and build the complete system, designers need to express 30 weave directives. When dealing with NF-concerns, we define 5 fragments, and use logical predicates to automatically identify 113 targets. The directive analysis step identify up to 40 shared join points in the system when the NF-concerns are included.

Modeled System	WEAVE	MERGE	Action*
Business-driven CCCMS	23	5	2422
Including NF concerns	86	40	10838

Table 11.1: Algorithms usage (& associated actions) when modeling the CCCMS

The final system (obtained after the execution of the composition directives) represents 1216 activities scheduled by 895 relations. All the models (inputs artifacts and composed ones) are

available on the case study web page² We summarize in TAB. 11.1 the number of times each algorithm is invoked and the number of actions performed on the initial model to execute the composition. Without specific optimizations, the composition engine implemented to support the ADORE approach consumes 68 seconds to build the final system (*i.e.*, main success scenario, business extensions and non-functional concerns).

Illustration. We use the use case *Capture Witness Report* to illustrate the design process. Based on the requirements and the designed artifacts, designers fill the invocation pool \mathbb{I} with the following directives³ I_n . These directives define a one-to-one mapping between the scenario extensions defined in the requirements and the ADORE algorithm invocation semantic.

$$\begin{aligned}
\clubsuit I_1 &= \text{WEAVE}(\{\omega(\text{callDisconnected}, \{a_{10}\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
\clubsuit I_2 &= \text{WEAVE}(\{\omega(\text{callDisconnected}, \{a_2\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
\clubsuit I_3 &= \text{WEAVE}(\{\omega(\text{requestVideo}, \{a_3, a_4\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
\clubsuit I_4 &= \text{WEAVE}(\{\omega(\text{ignoreDisconnection}, \{a_4\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
\clubsuit I_5 &= \text{WEAVE}(\{\omega(\text{fakeWitnessInfo}, \{a_2 a_3\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
\clubsuit I_6 &= \text{WEAVE}(\{\omega(\text{fakeCrisisDetected}, \{a_4\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
\clubsuit I_7 &= \text{WEAVE}(\{\omega(\text{fakeCrisisDetected}, \{a_3\}, \equiv)\}, \text{requestVideo}) \in \mathbb{I}
\end{aligned}$$

Based on this invocation pool, the *algorithm scheduler* identifies the two following situations: (i) two fragments are used for multiple integrations and must be cloned (*callDisconnected* & *fakeCrisisDetected*), and (ii) a shared join point exists (a_4). Based on these informations, the invocation pool is rewritten as the following (we use Greek letters to denote artifacts generated by the rewrite step) to handle the underlying CLONEPROCESS and MERGE invocations.

$$\begin{aligned}
I'_1 &= \text{CLONEPROCESS}(\text{callDisconnected}, \pi_1) \in \mathbb{I} \\
I'_2 &= \text{CLONEPROCESS}(\text{fakeCrisisDetected}, \pi_2) \in \mathbb{I} \\
I'_3 &= \text{MERGE}(\{\text{ignoreDisconnection}, \text{fakeCrisisDetected}\}, \mu_1, \equiv) \in \mathbb{I} \\
I'_4 &= \text{WEAVE}(\{\omega(\text{callDisconnected}, \{a_{10}\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
I'_5 &= \text{WEAVE}(\{\omega(\pi_1, \{a_2\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
I'_6 &= \text{WEAVE}(\{\omega(\text{requestVideo}, \{a_3, a_4\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
I'_7 &= \text{WEAVE}(\{\omega(\mu, \{a_4\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
I'_8 &= \text{WEAVE}(\{\omega(\text{fakeWitnessInfo}, \{a_2 a_3\}, \equiv)\}, \text{captureWitnessReport}) \in \mathbb{I} \\
I'_9 &= \text{WEAVE}(\{\omega(\pi_2, \{a_3\}, \equiv)\}, \text{requestVideo}) \in \mathbb{I}
\end{aligned}$$

The execution scheduling step identifies several dependencies inside \mathbb{I} . For example, the invocation I'_5 uses as input a cloned fragment produced by I'_1 . Based on a topological sort of the invocation according to these dependencies, the invocation pool is executed as the following (we denote as u_0 the initial universe):

$$\begin{aligned}
\text{Let } u_0 &\in \mathcal{U} \\
u_1 &= do^+(I'_1 \dot{\leftarrow} I'_2 \dot{\leftarrow} I'_3, u_0) \\
u_2 &= do^+(I'_9, u_1) \\
u_3 &= do^+(I'_4 \dot{\leftarrow} I'_5 \dot{\leftarrow} I'_6 \dot{\leftarrow} I'_7 \dot{\leftarrow} I'_8, u_2)
\end{aligned}$$

11.4.2 MERGE to Assess Shared Join Points

The MERGE algorithm aims to support designers while handling shared join points. We saw in the previous section that the CCCMS defines up to 40 shared join points when the NF-concerns are included. We identify two interesting situations in the context of the CCCMS which strengthen

²<http://www.adore-design.org/doku/examples/cccms/>

³At the implementation level, we provide a domain-specific language to make it more easy to express. See APP. A for more details about the current implementation of ADORE.

Merged Fragments Sets	Uses
$\{authenticateWhenIdle, logUpdate\}$	13
$\{logCreate, logTime\}$	4
$\{logCreate, logError, logUpdate\}$	2
$\{authenticateWhenIdle, logUpdate_{v_1}, logUpdate_{v_2}\}$	2
$\{logUpdate_{v_1}, logUpdate_{v_2}\}$	2
$\{logUpdate, logTime\}$	2
$\{logCreate, logUpdate_{v_1}, logUpdate_{v_2}, timeout, useHelicopter\}$	1
... 14 others combinations ...	1

Table 11.2: Fragments sets used as MERGE inputs in the CCCMS

the need for such a mechanism: (i) a merged fragment is re-used up to 13 times in the system and (ii) a shared join point consumes 5 fragments simultaneously. We summarize in TAB. 11.2 the most reused merged fragments in the context of the CCCMS.

11.4.3 WEAVE to Support Complex Behavior Definition

When the NF-concerns are included in the CCCMS, the WEAVE algorithm is used up to 86 times across the 12 initial fragments. We represent in FIG. 11.4 the *Capture Witness Report* use case, after composition⁴. We use the short notation to represent activity, and each color represents a fragment. We propose in the next section a in-depth analysis of the complexity of the CCCMS, based on quantitative software metrics.

11.4.4 Interference Detection Mechanisms in Action

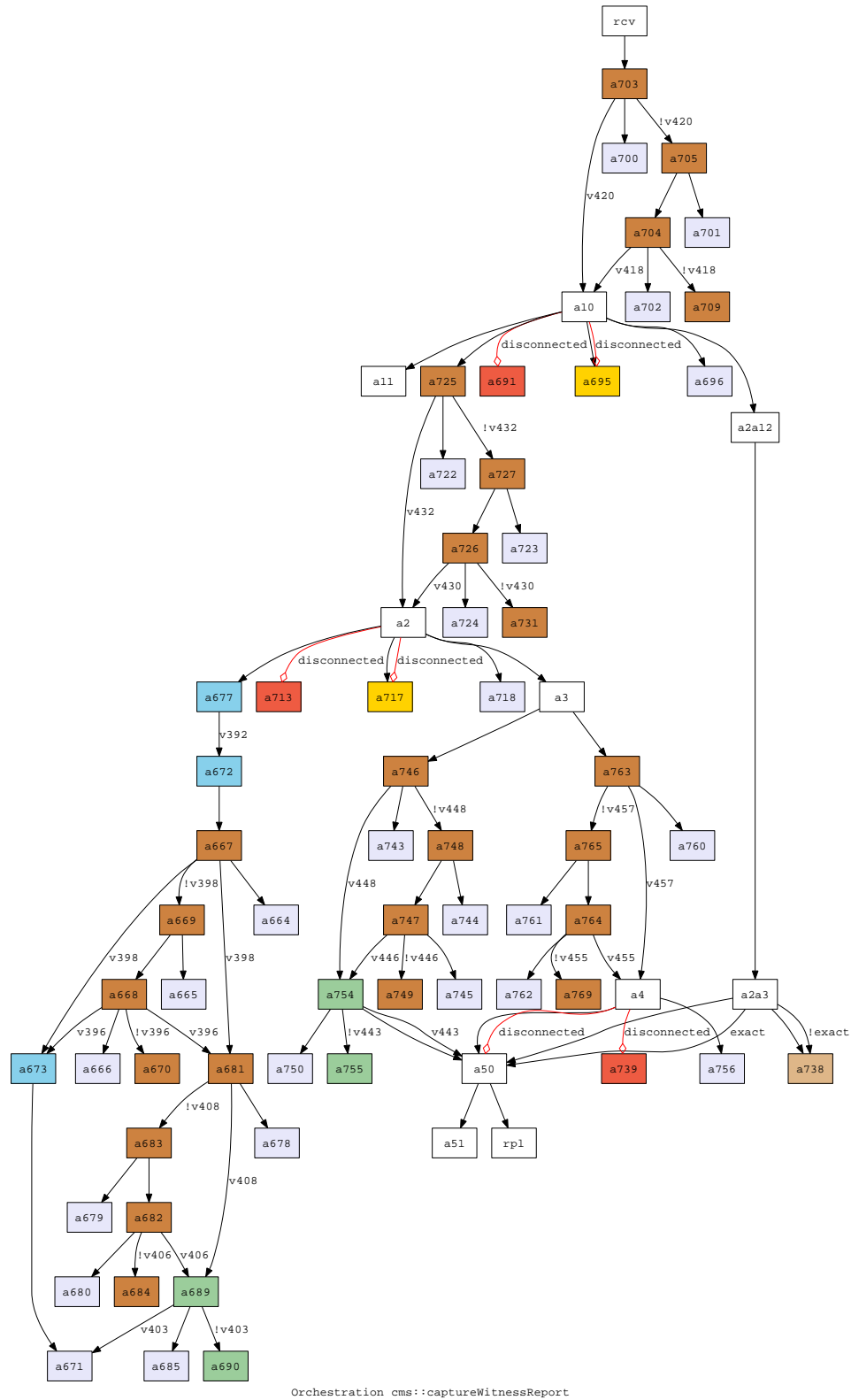
The large size and conceptual complexity of applications such as the CCCMS is the main motivation for using aspect-oriented orchestration modeling approaches such as ADORE. Separation of concerns reduces the complexity when focusing (locally) on one or a few fragments, but, at the same time, increases complexity when looking (globally) at the model as a whole. To tame this complexity we proposed to analyze the fragments and composed models to detect inconsistencies and bad-smells.

We use two sets of rules in the CCCMS, which allow us to identify lacks in the requirements document. The first ones rely on the separation of concerns paradigm, and identify troubles introduced by the composition itself. The second ones, more surprisingly, are only technical, and look for uninitialized variable or uncaught exceptions.

ADORE specific rules: Separation of Concerns. These rules are defined by ADORE in SEC. 7.3 to automatically identify “bad” situations introduced by the use of a separation of concerns approach.

- *Concurrent Termination.* This rule checks whether an ADORE model is deterministic or not. A deterministic ADORE model has only one well-defined response activity for one path in the activity graph. In the *Capture Witness Report* use case (#2), main success scenario failures are defined in extensions. For example, scenario extensions #5a and #5b are defined *in parallel*. The first one defines a failure when the *Phone Company* actor cannot verify witness informations. The second one defines another failure when the *Coordinator* actor declares this crisis as a fake one. Since those two different situations are handled by two different actors in parallel, the system is non-deterministic at the requirements level. The ADORE framework identifies this situation through the violation of this rule.

⁴The CCCMS intensively relies on a message bus where notification are sent to share information and status while handling a crisis. As a consequence, we relax the *activity path completeness* property (PROP. 4.16) for the notification activities. By relaxing this constraint, (i) we obtain more “compact” processes for representation purpose, and (ii) it does not impact the compositional properties according to the intrinsic “notification” intention of these activities.

Figure 11.4: *CaptureWitnessReport* use case (main success, extensions & NF-concerns)

- *Equivalent Activity Introduction.* Multiple fragments can introduce service invocations that are equivalent by weaving a same fragment on several activities or by requesting a same service in different fragments. A rule that brings these equivalent services to the attention of the modeler can cause the modeler to consider how the model can be refactored to avoid unnecessary redundant invocations of services. To illustrate this rule, we focus on the *statistics logging* NF-concern. To record statistical information on response time of each rescue team, we weave the fragment *logTime* (FIG. 11.3) around each invocation activity involving a `cmsEmployee`. When this fragment is woven on two immediately consecutive activities, it generates several activities which compute the same time (for example activities a_{746} and a_{763} in FIG. 11.4). The business process can be refactored by unifying these activities. A knowledge is added in the system to optimize the orchestrations involving these useless concurrency calls (this pattern was detected three times in the CCCMS). Another interesting illustration can be found in the *Capture Witness Report* use case. This rule has also detected that the *Coordinator* can inform the system that the crisis is a fake one by analyzing witness answers, or by looking at the video feed, if available. Consequently, the *Coordinator*'s approval will be requested twice when a video feed is available. In this case, we consider that the redundancy of the situation is handled by the `ui` service: it will not broadcast to the coordinator the same question multiple times.
- *Forgotten Weave.* When a fragment f is woven on an activity a' that is equivalent to another activity a that is not yet woven with f , ADORE displays a suggestion that the modeler may have forgotten to weave f on the activity a . As an example, we consider the use case #1 (*Resolve Crisis*)⁵. An extension (#5a) defines the process behavior when there is no available external resource. But extension #4a requests an external resource when there is no available internal resource. As a consequence, the error processing mechanism defined in extension #5a must also be triggered when extension #4a is used. This situation is detected by ADORE which informs designers of the situation.

Syntactical checks detect requirements lacks. ADORE defines syntactical checks which are too technical to be fully described in this document. For example, ADORE refuses the usage of an uninitialized variable as input of an activity. We found interesting that such technical checks allow us to identify lacks and weaknesses in the requirements document.

- *Uninitialized Variables.* This rule checks whether a variable is used before being initialized. Ensuring this property before composition is not sufficient to ensure it after composition: faulty interactions can lead to a composition result that violates this rule. Such a situation is typical when handling error in an orchestration by reconnecting the error flow with the normal one. Following the requirement document, in use case #1, each worker *must* submit a report to the system when the asked task is ended. But in extension 5a, which describes how the lack of an external resource is handled, there is no information on how to deal with the missing report. Consequently, the *report* variable may be uninitialized in the process. This situation is identified by ADORE.
- *Exception Handling.* This rule checks if the fault thrown by a process are handled in the caller process. Fixing such weaknesses is not mandatory for the designer who can decide to ignore the violations. In the *Resolve Crisis* use case, step 1 requests that the *Coordinator* captures the witness report. This step refers to use case #2 (*Capture Witness Report*), which can fail when a fake crisis is detected. The weakness is that there is no extension defining how the system should manage witness report when a fake crisis is detected. Another example is given by the NF-concerns. Weaving NF-fragments adds error throwing activities. But in the requirements document there is no information how to deal with these errors. This rule has identified this set of missing information in the requirements.

⁵The intention of the *Coordinator* actor is to resolve a car crash crisis by asking employees and external workers to execute appropriate missions.

11.5 Quantitative Analysis

In this section, we highlight the need of separation of concerns when designing a large set of business processes. We collect quantitative data (obtained from ADORE) representing the initial system and then make a comparison with data collected after composition (with or without NF-fragments). We use a set of software metrics inspired by state-of-the-art research about business process quality [Vanderfesten et al., 2007]. The analysis we present in this section is based on data that are relevant only in the context of the AOM case study. The raw data for metrics computation are available on the case study web site, and the complete quantitative analysis is publicly available⁶.

11.5.1 Coarse-Grained Structural Complexity

Definition. Software engineering community usually uses LOC (“lines of code”) to measure coarse-grained software size. As the LOC metric is not directly suitable when dealing with business processes we translate this coarse grained complexity using the activity set cardinality (denoted as $|\mathcal{A}^*|$).

Results. Our results according to this metric for the CCCMS case study are depicted in Fig. 11.5. We represent for each process $p/inprocs(u)$ the associated $|\mathcal{A}^*(p)|$, and compare three different versions of the CCCMS: (i) the initial system (main success scenario), (ii) the business-only system (including the scenarios extensions) and (iii) the final system (including both business extensions and non-functional concerns). The cardinality set average is multiplied by five between the initial system (7.83 activities in average) and the final one (39.25 activities in average).

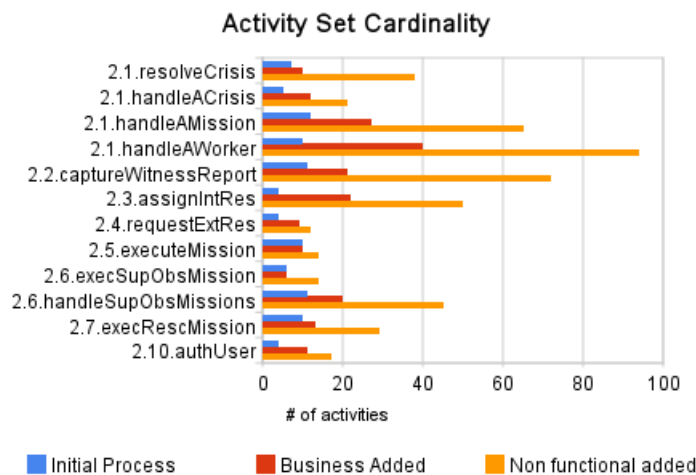


Figure 11.5: Evolution of the $|\mathcal{A}^*(p)|$ indicator

Analysis. This chart clearly shows that the number of activities involved in the CCCMS realization grows really fast. Talking about the evolution cost, up to 10.000 actions need to be performed on the initial models to build the final ones. The composition algorithm takes in charge the complexity of building the complete process. In the next section, we focus on this induced complexity by looking at the provenance of entities inside computed processes.

⁶<http://spreadsheets.google.com/pub?key=tgS6qbzqo5CxTxlcTXbtsPA>

11.5.2 Entity Provenance

Definition. This section is a corollary of the previous one. Based on the coarse-grained complexity of process, we identify how many activities came from the initial orchestration, the business fragments and finally the non-functional fragments. This indicator allows us to quantitatively qualify which part of the final system was initially defined in the requirements. We normalized these values using the final cardinality ($|acts(p)|$) to obtain an activity provenance percentage.

Results. Figure 11.6 represents these indicators for activities in the context of the CCCMS. We can immediately notice that in average, more than 50% of a final process is defined as non-functional activities. Extrema values are interesting too: the **assignIntRes** (use case #3) process contains only 8% of initial business activities. On the contrary the **execRescMission** process (use case #7) is composed of more than 70% of initial activities. These values conform to the requirement documents, as scenario #3 defines only two steps in the main scenario and nine in its extensions. On the contrary, scenario #7 defines seven steps and only three in its extension.

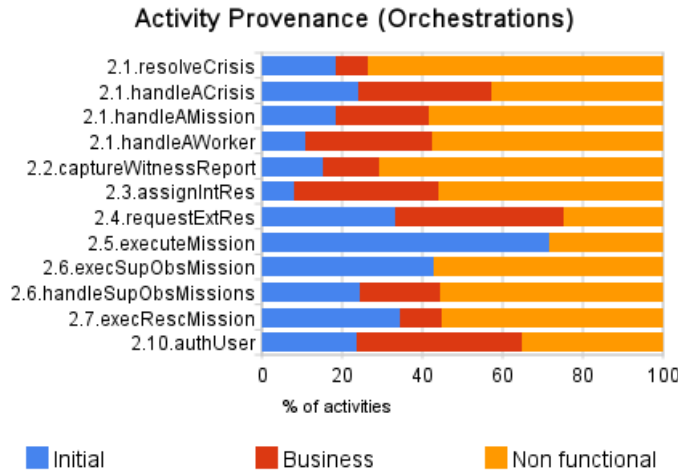


Figure 11.6: Activities provenance in the final CCCMS

Analysis. These indicators enforce the straightforward mapping between textual use cases and designed process. A large process with small extensions will be defined as a large orchestration and few fragments. But it also demonstrates the need of automatic composition, as in some case up to 75% of a process is defined as extensions of a main scenario.

11.5.3 Cognitive Load

Definition. The cognitive load indicator aims to quantify the intrinsic complexity of a business process. It is defined as an coarse grained approximation of the Control Flow Complexity indicator [Cardoso, 2005], based on two simple concepts: the process *surface* and the *labyrinthine* complexity. *Surface* is computed as a product between process *width* (i.e. number of activities executed in parallel) and process *height* (i.e. longest path between an entry point and a exit point). The *labyrinthine* complexity represents the number of different paths available in the process. Inspired by [Laue and Gruhn, 2006] who defines cognitive load of programs as a linear function (based on structural complexity), we define the cognitive load of an ADORE process as the multiplication of its surface and its labyrinthine complexity, normalized by the number of activities inside the process ($|acts(p)|$). This indicator allows us to apprehend both *computational* and *psychological* complexities as defined in [Cardoso et al., 2006].

$$CognitiveLoad(p) = \frac{Surface(p) \times Labyrinthine(p)}{|acts(p)|}$$

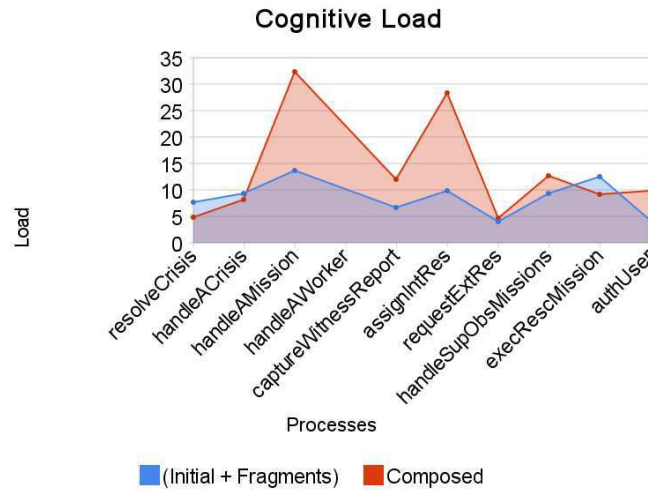


Figure 11.7: Cognitive load indicator (Main Success Scenario + Business Extensions)

Results. We focus in this part on the business-driven CCCMS, *i.e.*, the initial system and its business extensions⁷. For each process (excepting the `handleAWorker` one, as its cognitive load is 247.5), Figure 11.7 represents the sum of initial processes and used fragments cognitive load, and the cognitive load of the final process. For five processes (50%), the final load is clearly higher than the sum of initial process and used fragments loads. For the other processes, the final cognitive load follows the same magnitude than the cumulated one.

Analysis. This indicator clearly illustrates the immediate advantage of separation of concerns to tame the complexity of designed artifacts. But it also highlights one of the weakness of the ADORE platform (and more generally the AOM approach). Designing small processes (or process without multiple extensions) using the separation of concerns paradigm may introduce an overload in the design process. This overload is not visible in terms of result, but can be summarized in the following sentence: “*When should one write several small concerns and then express a composition when he/she can directly express the expected result ?*”. Such a typical useless composition is illustrated in use case *Execute Rescue Mission* (#7), as the main scenario is implicitly designed to handle the sole extension defined on it: a request is sent in the main scenario, and the response to this request is handled inside the extension.

11.6 Conclusions

11.6.1 Approach Intrinsic Limitations

Abstraction at workflow level. ADORE aims to support separation of concerns at the level of workflows. The introduction of new concerns inside an activity is not managed by the ADORE framework. In other words, we consider atomic activities and services as black boxes and do not provide any mechanisms to enhance them internally. Moreover, when designing such a big

⁷We consider than non-functional concerns should not be handled manually in an AOM approach

system as CCCMS, other separation of concerns features are needed to design structural information about services and data. This is another limitation of the ADORE platform and one of our perspectives.

Visualization scalability. The previous section demonstrated the scalability of the approach in the CCCMS context. The possibility to visualize and analyse partial composition such as behavioral merge results helps taming the complexity of business processes design. One of the limitations of ADORE is to provide visualizations only for separate entities: when a system involves many processes, it is necessary to have a holistic point of view on compositions and business processes to grasp it. So other visualization methods are needed to tackle complexity of compositions at the global system level. We start a collaboration with the MONDRIAN (a visualization tool) community to address this problem [Mosser et al., 2010a]. The definition of accurate visualization techniques is an ongoing perspective.

Pointcut abstraction level. The ADORE framework does not support complex pointcut definition using a surface language. As soon as we need quantification we need to use directly the logical back-end. This approach is powerful and supports almost all kind of quantification. But the cost is a definitive loss of abstraction mechanisms. We do not consider it as an insurmountable drawback since business process modelers normally use explicit targets when they use process indicators.

11.6.2 Summary

This chapter describes how to use the ADORE method to model the Crisis Management System Case Study as a SOA. In the approach, use cases are modeled as service orchestrations. For each use case, the main scenario is realized as a base orchestration, and the use case extensions and non-functional properties are modeled as fragments. All the designed models are available on the associated website⁸.

A quantitative analysis of the approach has been performed based on a set of metrics inspired by state-of-the-art research on measuring business process complexity. It demonstrates the benefits of separation of concerns to tame the complexity of creating and evolving models of large business processes in which many functional and non-functional concerns must be addressed. It also established that automatic merging and weaving of fragments are essential to deal with this complexity. The case study provides some evidences that the ADORE method is scalable. The orchestrations produced by the compositions in the case study are moderately large and complex as indicated by the size metrics we've gathered. Introducing non-functional concerns complexify the processes in a really deep way. The automated composition shields the developer from the complexity of composing orchestrations.

From a utility standpoint, we were able to realize all the use cases without encountering any methodological problems. ADORE intrinsically supports evolution of base orchestrations. We illustrate this feature in the context of this case study by considering each scenario extension as an evolution of the associated main success scenario. The introduction of non-functional properties that impacts processes behaviors such as persistence or statistical logging follows the same methodology. The DATAFLOW algorithm was successfully used to introduce iterations over available resources while handling the crisis.

An important ADORE goal is to support end-user design of orchestrations, where an end-user is a business process modeler. As a consequence, the modeling language utilizes concepts and expressions that should be familiar to business process modelers. Once a process modeler has specified where fragments are to be woven into the base model, ADORE composes the fragments and base models automatically and provides indications of possible problems arising out of interactions between fragments and base orchestrations.

Some interference rules only identify design *bad-smells*, but others detect errors that make the composed result inconsistent. Such inconsistencies are detected and exposed to the process modeler, who can then fix it. As a consequence, we do not ensure an *a-priori* orchestration

⁸<http://www.adore-design.org/doku/examples/cccms/start>

correctness property. It is interesting to notice that these rules were able to detect requirement weaknesses in the context of this case study.

At the implementation level, the composition engine handles the composition of the complete application in 68 seconds, without any optimization on the PROLOG code. Moreover, we supervised an M.Sc. three weeks project where four students of the POLYTECH'NICE school had implemented in Java the processes computed by the composition engine. The implementation coverage is not complete, but it does not seem to trigger any issue in the ADORE approach, at the platform level.

JSEDUITE, an Information Broadcasting System

«It is a very sad thing that nowadays there is so little useless information.»

Oscar Wilde

Contents

12.1 Introduction	167
12.2 From Architectural Principles to Implementation	168
12.2.1 Information Broadcasting Candidates (IBC)	168
12.2.2 Profile & Sources of Information	168
12.2.3 Information Providers	170
12.2.4 Implementation	170
12.3 Capitalizing Designers Knowledge with ADORE	170
12.3.1 Supporting Providers & Sources Design (P_1, P'_1)	171
12.3.2 Broadcasting Policies & NF–Concerns as Fragment (P_2, P'_2)	171
12.3.3 Multiple Calls as DATAFLOW Usage	173
12.4 Conclusions	175
12.4.1 Summary	175
12.4.2 Towards an Information Broadcast Software Product Line	175

12.1 Introduction

This chapter describes the second case study of ADORE capabilities. It aims to explain the usage of the framework to capitalize architect best practices and help orchestration definition. Contrarily to the CCCMS which was a model-driven case study, this chapter focuses on the re-engineering of an existing piece of software. A preliminary version of this work is described in [Mosser et al., 2008b, Mosser et al., 2009b].

JSEDUITE is an information system designed to fit academic institution needs. The initial version and first proof of concept of the system was published in 2005. The current stable version was released in February 2010, and has involved eight contributors at the code level (with up to five concurrent work spreads over three continents). It supports information broadcasting from academic partners (e.g. transport network, school restaurant) to several devices (e.g. user's smart-phone, PDA, desktop, public screen). This system is built upon a WSOA and used as a validation platform by the FAROS project¹. The system is deployed inside three institutions (POLYTECH'SOPHIA engineering school & two institutions dedicated to visual impaired people: (i) CLÉMENT ADER institute dedicated to childhood and (ii) the IRSAM association for adult people.

¹<http://www.lifl.fr/faros> (French only)

This chapter is organized as the following: in SEC. 12.2, we describe the key concepts of JSEDUITE in terms of business processes. We describe in SEC. 12.3 how ADORE is used to tackle the issue encountered while designing the system manually. Finally, SEC. 12.4 concludes this section, and expresses an interesting perspective of the system (related to software products lines).

12.2 From Architectural Principles to Implementation

The JSEDUITE architecture is built upon three concepts at the business process level: (i) Information broadcasting candidates, (ii) sources of information and (iii) providers.

12.2.1 Information Broadcasting Candidates (IBC)

First of all, we define an information as a data exchanged inside the system and finally delivered to a broadcasting device. Such data can be instances of atomic types (e.g. `string`, `integer`, `date`) or more complex data structures (e.g., `RestaurantMenu`, `TeacherAbsence`).

We called *Information Broadcasting Candidates* (IBC) existing web services which deliver relevant information for academic institutions. The current implementation of JSEDUITE contains 13 IBC. These service are implemented as elementary services or orchestrations of services. Elementary services include dedicated realizations (e.g., restaurant menu database) or preexisting services provided by partners (e.g., RSS reader, timetable hosting). The orchestrations of services used to implement IBC aggregate elementary operations to provide an added-value service. The PICWEB example is one of these orchestrations. We represent in FIG. 12.1 the available IBC.

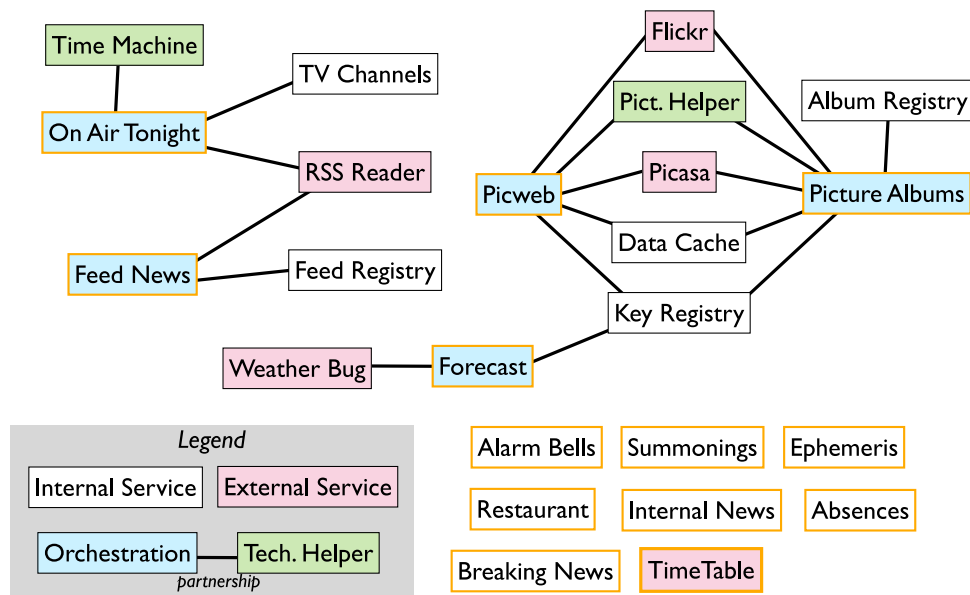


Figure 12.1: Available Information Broadcasting Candidates (orange) in JSEDUITE

12.2.2 Profile & Sources of Information

The previous section defines IBC as regular web services. As a consequence, the operations one can use to retrieve information from a given IBC accept a wide range of parameters. For example, the PICWEB orchestration consumes a *tag* and a *threshold* to produce the relevant set of pictures to be broadcasted in the system.

The JSEDUITE system tackles this problem by reifying a *profile* concept. A profile is uniquely identified in the system, and associates parameters name and value. Based on a given profile,

one can query it and extract parameters values associated to an IBC. For example, in the POLYTECH’NICE engineering school, we defined two profiles: (i) the *public* profile asks PICWEB to look for pictures with the *polytech* tag, and (ii) the *studentCommitee* profile retrieves pictures published by the students associations. This mechanism allows school headmasters to dispatch the information flow according to different assistances.

Based on this concept of profile, we define a *source* as a mediation layer between an IBC and the JSEDUITE system. Each IBC is encapsulated by a *source*, which performs the mediation between real-life parameters and the JSEDUITE profile system. This mediation layer allows one to handle any IBC through a common interface.

We implement a source as an orchestration of services which realizes a *draw* operation. A source receives a profile identifier as input, and computes the relevant set of information associated to its associated IBC. We depict in FIG. 12.2 using the ADORE graphical syntax the source associated to the PICWEB candidate. The source starts by asking the profile **manager** to know if its associated IBC needs to be invoked (activity *test*). If not, an empty list of information is returned (*ign₁*, *ign₂*). In the other case, the **manager** is invoked (*par₁*) to retrieve the parameters defined in the profile for this source. The values are then extracted (*par₂*, *par₃*), before performing the invocation of the real service (*call₁*). The retrieved data are transformed into a list of information (*call₂*), and finally replied to the caller (*rpl*).

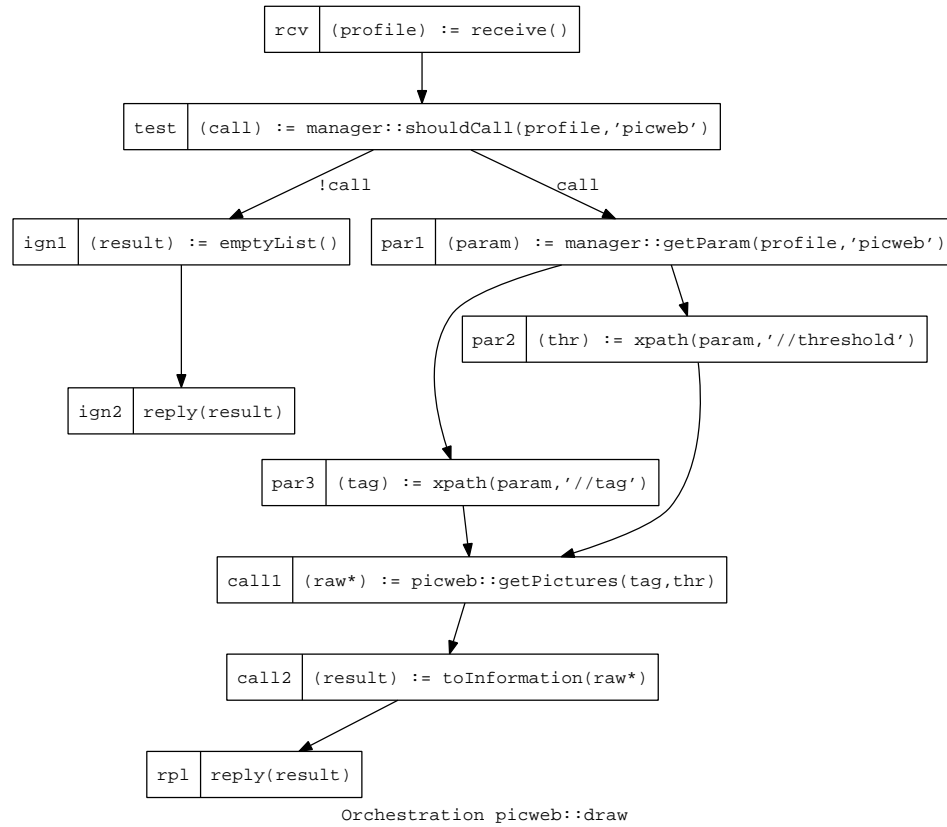


Figure 12.2: Source associated to the PICWEB Information Broadcasting Candidate

Problems. Defining new sources is a repetitive task (P_1). At the implementation level, this work is done with an intensive usage of the *copy/paste* anti-pattern, a new source deriving from the previous one. Several sources hold *broadcasting policies*, which modify the way they interact with their associated IBC. For example, the **RestaurantMenu** source contains a piece of code which dries it up after lunch time. Even if documented, these policies are *interpreted* by

designers, and re-applied in a slightly different way in another source (P_2). Such a development process jeopardizes the global maintenance of the set of business processes defined in the system.

12.2.3 Information Providers

Based on the previously defined sources, we implement *information providers* as the keystones of the JSEDUITE system. A *provider* aggregates several sources of information, by invoking them and merging the associated sets of information into a single one. This set is manipulated according to the provider target. For example, a provider designed for mobile devices such as smartphone will lighten information size by removing non-relevant details (such as large images). Each broadcasting device is connected to a provider. We represent in FIG. 12.3 an overview of this architecture. Providers allow to restrict the scope of sources: a “public” provider (connected to a large screen in the institution lobby) will be only connected to “publicly available” sources of information.

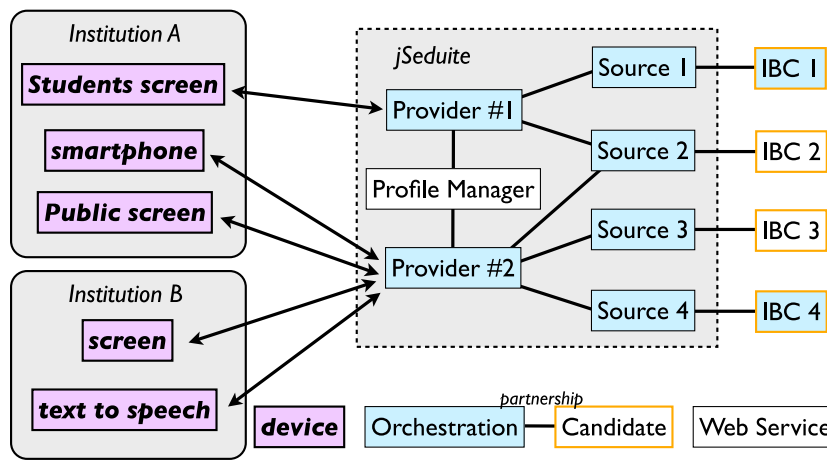


Figure 12.3: JSEDUITE big picture: Information Providers, Sources and IBC

Problems. Providers suffer from the same issue than the sources of information. Even if the process of defining a new provider is well-documented in the project, designers use *copy/paste* to derive a provider from another one (P'_1), and then interpret the broadcasting policies before re-implementing it (P'_2). Non-functional properties (e.g., authentication, cache mechanisms) are manually inserted too.

12.2.4 Implementation

At the implementation level, JSEDUITE is composed by 23 web services (written in JAVA with the JAX-WS stack), 7 business processes (written in BPEL, WSDL and XSD). It represents approximately 70,000 lines of code (and $\sim 125,000$ lines of code when including generated XML artifacts). We identify 13 IBC, and then implement the 13 associated sources of information. Based on these sources, we implement 4 providers. JSEDUITE is published using the LGPL license and benefits of an APP² registration.

12.3 Capitalizing Designers Knowledge with ADORE

We describe in the previous section the JSEDUITE architecture and its associated implementation. We also describe several issues identified while developing JSEDUITE. We describe in this

²“Agence pour la Protection des Programmes”, French institution for software ownership protection

section how ADORE is used to handle such problems, by capitalizing designers knowledge into reusable fragments.

12.3.1 Supporting Providers & Sources Design (P_1, P'_1)

We describe in the previous section how the manual design of a provider (or a source) is an error-prone and time-consuming process. We propose to use a compositional approach to tackle these issues. Using the ADORE formalism, we define an initial *empty* provider (FIG. 12.4(a)), which plays the role of the initial system (basically doing nothing). We also define the *addSource* fragment (FIG. 12.4(b)), which describes how a source will be inserted inside *empty*: The new source is invoked in parallel of the initial behavior (a_0), and the retrieved data are concatenated with the legacy ones.

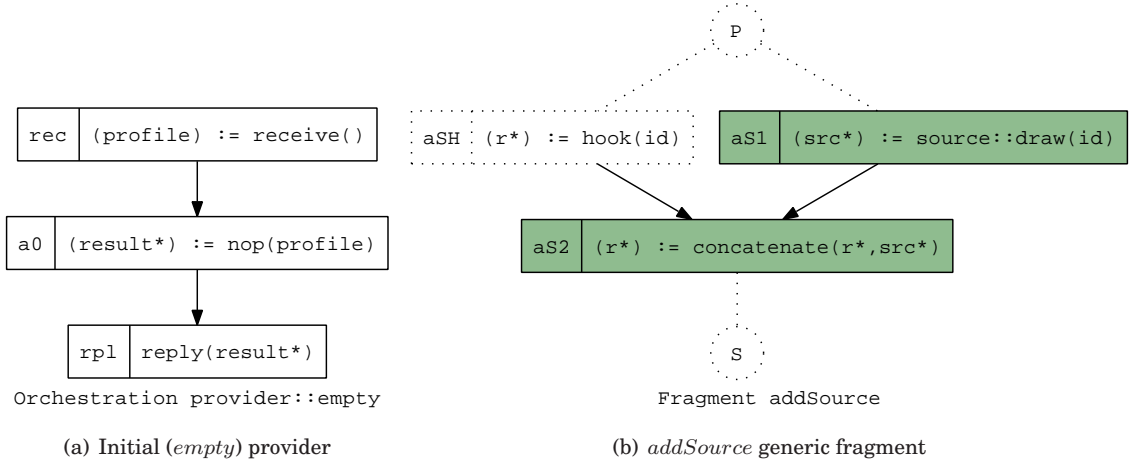


Figure 12.4: Initial artifacts designed to support providers definition

Based on these artifacts, the complexity of designing a new provider is shifted into the expression of the associated composition directive. We consider now two designers *Bob* and *Alice* working together on the same provider called *sample*. *Bob* works with the *feedNews* source, and *Alice* handles the *picweb* one. They simply need to (i) clone the *addSource* fragment to obtain their own copy, (ii) substitute the invoked source by the expected one and finally drop off a weave directive in the invocation pool.

$$\begin{aligned}
 \Leftarrow I_1 &= \text{WEAVE}(\{\omega(\text{feedNews}, \{a_0\}, \equiv)\}, \text{empty}) \\
 \Leftarrow I_2 &= \text{WEAVE}(\{\omega(\text{picweb}, \{a_0\}, \equiv)\}, \text{empty})
 \end{aligned}$$

The algorithm scheduler identifies a shared join points, and triggers a MERGE algorithm. At the merged framgent level, an interference is detected: the r^* is concurrently accessed by the two concatenation activities (derived from aS_2). However, in the context of JSEDUIE, a domain-dedicated knowledge states that when two concatenation interfere, the engine can choose an arbitrary order to solve it. As a consequence, the composition supports provided by ADORE automatically build the process depicted in FIG. 12.5 (*Bob* is blue, *Alice* is yellow).

The same pattern is followed to support the design of new sources. The only variability dimension between two sources is the number and the name of parameters retrieved in the profile. Consequently, we define a generic business process *abstractSource*, and a fragment *readParameter*. Based on these two artifacts, one can generate any kind of sources.

12.3.2 Broadcasting Policies & NF-Concerns as Fragment (P_2, P'_2)

Like for the CCCMS system, we do not make any difference between functional and non-functional concerns when designing fragments in ADORE. We identify in the system eight broadcasting poli-

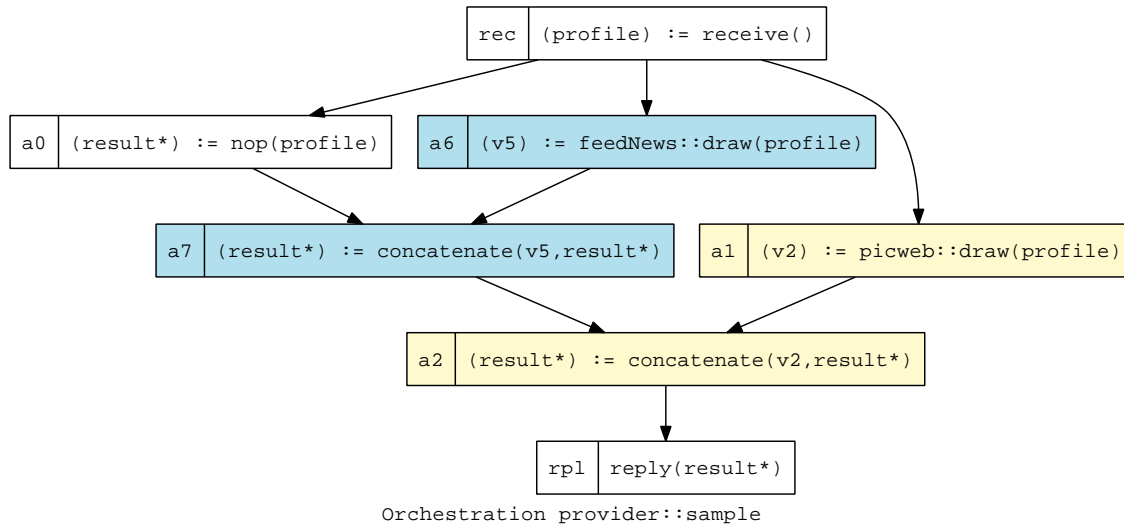


Figure 12.5: Automatically built provider, aggregating *feedNews* and *picweb*

cies and two non-functional concerns. These ten concerns are factorized as fragment of processes. For concisions reasons, we do not graphically represent each fragment, and simply summarize its behavior in a few lines. The associated implementation artifacts are available on the ADORE website:

http://code.google.com/p/adore/source/browse/trunk/case_studies/jseduite

Broadcasting Policies. We design the eight broadcasting policies identified in JSEDUITE as fragments.

- *diet*. The fragment sends the hooked set of information to a dedicated service able to delete heavy details. This is particularly useful when the information needs to be broadcasted to mobile device, due to the restricted bandwidth. For example, a **News** contains information about its author, a short title, a summary, a set of tags, a full length content and a list of associated news URLs. In this situation, we delete the two last items to reduce the size of the data when exchanged over the network.
- *dry*. Several types of information do not make sense after (or before) a given hour. This fragment asks for the system time and performs a comparison between the current time and the one contained in the fragment. If the broadcasting condition is not satisfied (*i.e.*, it is too late or too early), it returns an empty list of information. For example, the **Restaurant-Menu** source must be dried up after 2PM in the POLYTECH’NICE school (there is no dinner service in this restaurant). The opposite example is the **OnAirTonight** source (retrieving the evening TV shows schedule), which should not be broadcasted before the late afternoon to let school users focus on their daily work.
- *filter*. This fragment filters a set of information according to the decision given by an oracle service. School headmasters can then restrict the scope of some information. Using this fragment, one can decide to delete **News** handling the “*internal only*” tag from the publicly available providers.
- *inhibitsOnBreaks*. JSEDUITE feeds internal plasma screen to broadcast publicly available information such as fellow students timetable. When a break (*i.e.*, a short time between two lectures) happens, it implies that a large amount of school users will walk in front of the screens. One may want to inhibit several sources of information to only broadcast very important information (*e.g.*, next lecture location, student summons) during breaks. A

fragment (a more complex version of the *dry* one) is defined to handle such a situation. For example, we decide to only broadcast next lecture location and internal news during breaks in POLYTECH'NICE.

- *reorder*. This fragment sends the retrieved set of information to an oracle service which transforms it into an ordered list of information, according to information priority. It allows headmasters to organize their information at different level of priority. Such a mechanism is needed by the IRSAM association, which accommodates people with mental handicap. Important information (*e.g.*, doctor's name and location) must be broadcasted just after the screen wait page due to their mental weak memory capabilities.
- *resize*. The resize fragment is dedicated to picture-driven information. It analyzes the retrieved pictures, and sends the unnecessarily large entities to a resize service. This service resizes the pictures and publishes their resized version on the local network. This mechanism reduces the network bandwidth usage, but may slow down the global response time (image analysis are costly operations). Consequently, it is often combined with a *cache* (see NF-concerns) for performance reasons.
- *shuffle*. Considering a set of information as an ordered list (which is the underlying data structure, relying on XML sequences), this fragment blends it to change the internal order. This fragment is often used in conjunction with the *truncate* one to generate pseudo-random informations (*i.e.*, the retrieved set is shuffled and then truncated).
- *truncate*. The truncate service was described in the PICWEB running example. This fragment hooks a set of information and sends it to the *truncate* business process.

Non Functional Concerns. We identify two non-functional concerns in the system. These concerns are reified as fragment too.

- *authenticate*. We define a *token-driven* authentication mechanism to deal with access control list. This fragment hooks the reception of a profile identifier. It then asks the authentication server to check if the profile holds a token valid for the current process. If not, a security exception is thrown.
- *cache*. The cache fragment is equivalent to the one defined in PICWEB. The only difference is the way the *key* used to identify the stored data is computed. In this fragment, the key is computed as the concatenation of the profile identifier and the current process name.

“Points of Interest”. The previously defined fragments aim to be integrated into sources and providers. These two classes of processes define point of interest where the fragment will be woven: (i) *profile* reception, (ii) response sending and (iii) usual behavior (*i.e.*, IBC invocation for a source, initial empty behavior for a provider). The “usual behavior” point is used by the fragment which supports the construction of the artifact (*i.e.*, *addSource* and *readParameter*), and by the *cache* one. The *profile* reception is adequate to weave inhibition fragment (*e.g.*, *dry*). The response sending point allows the definition of information sets (*e.g.*, *shuffle*). We represent in TAB. 12.1 where the different fragments are intended to be used.

12.3.3 Multiple Calls as DATAFLOW Usage

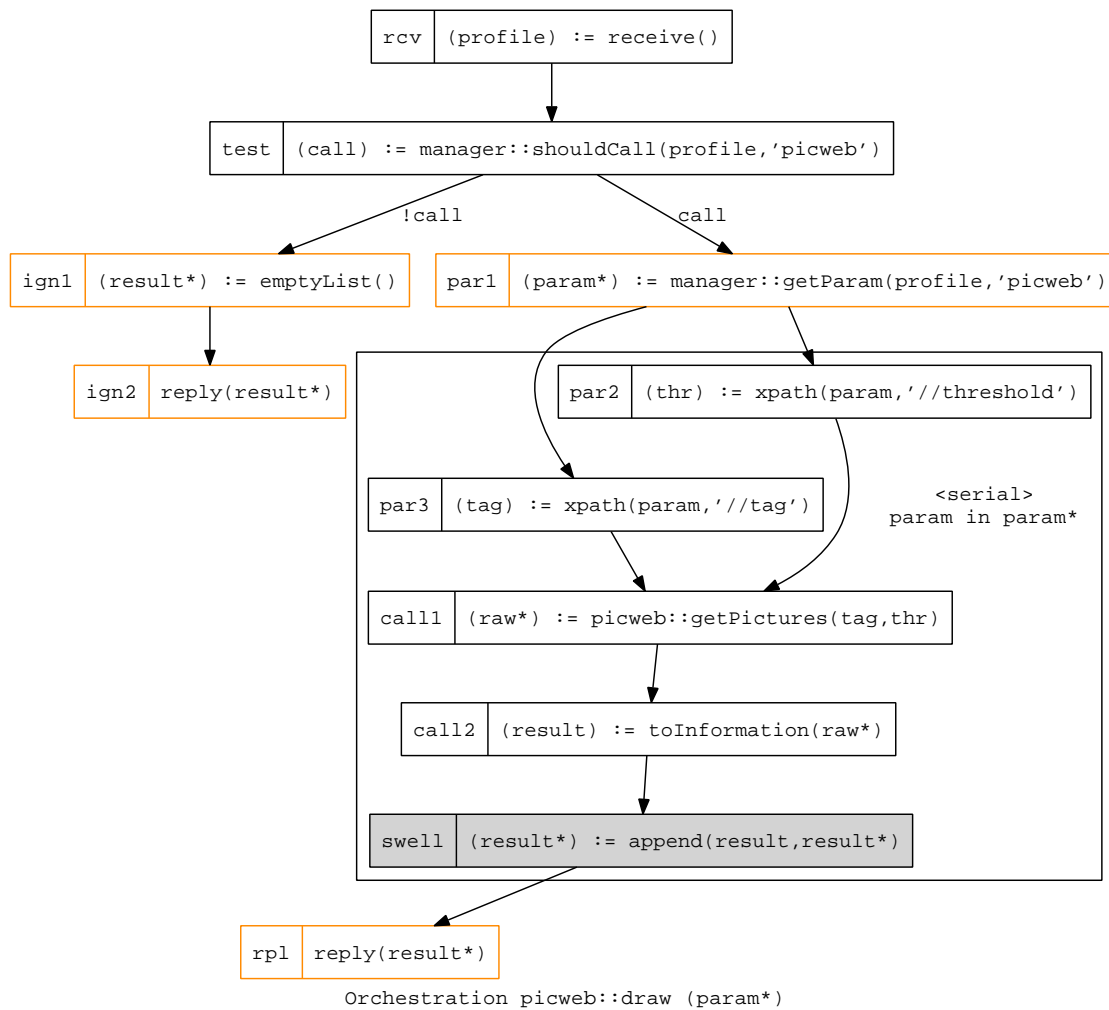
We describe in the next section how a source is designed. However, this design is simplistic, and do not allow a profile to ask for the same source multiple times. For example, the *timetable* IBC consumes the identifier of a lecture group, and returns the associated schedule. As a consequence, to retrieve multiple schedules, we need to store in the profile multiples group identifiers, and iterate over this set of parameters. This is exactly the goal of the DATAFLOW algorithm.

We depict in FIG. 12.6 a version of the *picweb* source able to handle a set of parameters, *i.e.*, a set of tags and thresholds information. In this figure, we highlight in orange the activities which are not part of the iteration, but still impacted by it. The activity *par*₁ receives now a set of parameters, and the activities *ign*₁, *ign*₂ and *rpl* work on list of depth 2. This example

Fragment	Target			Process Class	
	Rec.	Resp.	Usual	Source	Provider
<i>diet</i>		×		×	×
<i>dry</i>	×			×	×
<i>filter</i>		×			×
<i>inhibitsOnBreak</i>	×			×	
<i>reorder</i>		×			×
<i>resize</i>		×		×	×
<i>shuffle</i>		×		×	×
<i>truncate</i>		×		×	×
<i>authenticate</i>	×			×	
<i>cache</i>			×	×	×

Table 12.1: Broadcasting Policies as Fragment in JSEDUITE

strengthen the need for algorithm like the DATAFLOW one in the context of SOA: this simple change ($param \mapsto param^*$) creates an iteration containing four legacy activities, add a new activity and its associated relations, and modify four activities inside the process. Finally, only the two first activities where not impacted by the iteration introduction.

Figure 12.6: Automatic handling of multiple invocations in the *picweb* source

12.4 Conclusions

12.4.1 Summary

In this chapter, we described how ADORE is used to capitalize designers knowledge while designing SOA business processes. This knowledge is reified as fragment and then composed to build new architectures. The methodology is illustrated on a legacy application, daily used in several institutions.

Based on two initial orchestrations and two enhancement fragments, we are able to build any artifact needed by the existing JSEDUITE system (*i.e.*, *source* or *provider*). Moreover, we reify broadcasting policies into reusable fragments of processes, which can be automatically integrated into legacy process on an on-demand basis.

One limitation of the method is the knowledge basis defined by ADORE. Its expressiveness is restricted at the instance level and does not allow one to express business-driven knowledge such as “*Always resize after truncating*” of “*Choose an arbitrary order between two parallel concatenations*”. We simulate this business-driven mechanism at the implementation level, but handling it properly at the model level is still an open issue.

12.4.2 Towards an Information Broadcast Software Product Line

The domain-dedication idea exposed in the previous paragraph leads to a more general idea. By capturing the business domain of information broadcasting into a meta-model, we can define a Software Product Line dedicated to information broadcasting, at the business-level. The idea is to target directly school headmasters. With such a product line, a headmaster will choose a subset of available sources of informations, select policies to be applied on these sources, and then build providers using his/her vocabulary. The underlying complexity of creating orchestrations, applying fragments and solving conflicts is then given to an automatic engine. This perspective is one of the short-term perspectives for the JSEDUITE system. Existing works on software products lines [Thaker et al., 2007, Freeman et al., 2008, Uzuncaova et al., 2010] will support the definition, the composition and the assessment of the system. As a consequence, a generative environment will be available for school headmasters, who will simply express configurations and then obtain as a result a customized system.

In this part, we described two large case studies used to assess the ADORE approach and its associated implementation in real-life context.

The CCCMS application is realized as an answer for a common case study on AOM. It aims to model the behavior of a Car Crash Crisis Management System. The requirements document was defined by external stakeholders, which emphasizes the fact that ADORE can be used in real-life context. This case study also shows the scalability of the approach and its implementation, considering the CCCMS as a very complex system, orchestrating of lot of services invocations. Moreover, interference detection mechanisms identify lacks in the requirements, at the process level. Finally, the ADORE approach allow us to cover all the use cases defined in the document, without any methodological issue.

The JSEDUITE application is based on a legacy implementation, daily used inside several academic institutions. This open-source system represents 70,000 lines of code. We identified several issues while developing the system, as repetitive, time-consuming and error-prone situations. We perform a reverse engineering of the JSEDUITE system, using ADORE as a compositional approach to support its design. The system obtained by composition fits our needs, and tackles all the issues identified in the initial development. Such an automated composition support lets us imagine the definition of a software product line associated to the system.

~> Based on these two case studies, we defend that ADORE fulfills its initial goal and then supports the definition of complex business processes using a separation of concerns approach. We describe in the next part perspectives of this work identified while realizing theses case studies, before concluding this dissertation.

Part V

Perspectives & Conclusions

Notice: This part ends this dissertation. We expose in CHAP. 13 exciting perspectives opened by this work. finally, CHAP. 14 concludes this dissertation by summarizing the contribution made with ADORE

Perspectives: Towards a Complete Development Approach

«Anybody who has been seriously engaged in scientific work of any kind realizes that over the entrance to the gates of the temple of science are written the words: “You must have faith”.»

Max Planck

Contents

13.1 Introduction	181
13.2 Upstream: From Requirements Engineering to ADORE	182
13.2.1 Context	182
13.2.2 Proposed Approach	182
13.2.3 Firsts Experiments	183
13.2.4 Summary	184
13.3 Downstream: Visualizing & Assessing Compositions	184
13.3.1 Context	184
13.3.2 Proposed Approach & Firsts Experimentations	184
13.3.3 Summary	186
13.4 Sketched Perspectives	186
13.4.1 Users Experiments	186
13.4.2 ADORE Expressiveness & Tooling	186
13.4.3 Actions Sequences & Optimizations	187
13.4.4 Knowledge Representation & Semantic Web	187
13.4.5 Dynamic Weaving	187

13.1 Introduction

Before concluding this thesis, we describe in this chapter several perspectives for the ADORE framework. References to these perspectives were spread in the document, where their motivating need was identified. We differentiate two kinds of perspectives: the first ones are based on ongoing research collaborations, and illustrate preliminary results interesting enough to be followed in a short term period. The second perspectives are only expressed as “sketch”, as we consider these directions interesting but our research on the associated domain is very prospective.

This chapter is organized as the following: SEC. 13.2 describes a methodological approach used to link a requirement engineering engine to ADORE. In SEC. 13.3, we propose a link between ADORE and a visualization engine, used to graphically assess the composed system. Based on these two collaborations, we tend to the definition of a complete model-driven business process development framework, handling requirements, design and assessment models. Finally, we describe in SEC. 13.4 sketched perspectives.

13.2 Upstream: From Requirements Engineering to ADORE

This work is the product of a collaboration initiated in 2010 with Aymot and Mussbacher, University of Ottawa [Mosser et al., 2011]. In this work, we propose to link ADORE with an upstream requirements engineering method and then automatically generate design artifacts (as ADORE models) based on requirements models.

13.2.1 Context

While designing a system, business process designers have to design workflows in their contextual environment, thus spending time on non-business oriented tasks to integrate non-functional concerns. Several adaptations may occur on the same workflow due to the many different concerns such as cost, efficiency, and security. These issues are tackled by the ADORE framework, at the model level. However, designers have to ensure that the workflows conform to the initial and adapted requirements.

Based on these observations, there is a need for helping users move from requirements to the design of accurate workflows while taking into account several possibly crosscutting concerns. Changes at requirements or design levels must be supported to apply adaptations in a safe way in both ways. Over the last decade, many aspect-oriented software development (AOSD) techniques [Chitchyan et al., 2005] have been proposed. Less attention has been paid to automatically link together these approaches across life-cycle phases, even though AOSD claims improved productivity when crosscutting concerns are encapsulated across these phases.

The Aspect-oriented User Requirements Notation (AoURN) [Mussbacher and Amyot, 2009, Mussbacher et al., 2010] supports the elicitation, analysis, specification, and validation of requirements in an aspect-oriented modeling framework for early requirements with the help of goal and scenario models. AoURN introduces concerns as first-class modeling elements, regardless of whether they are crosscutting or not. Typical concerns in the context of AoURN are stakeholders' intentions, non-functional requirements, and use cases. AoURN groups together all relevant properties of a concern such as goals, behavior, and structure, as well as pointcut expressions needed to apply new goal and scenario elements or to modify existing elements in the AoURN model. AoURN's scenario models are defined with Aspect-oriented Use Case Maps formalism (AoUCM)

13.2.2 Proposed Approach

We propose an iterative, concern-driven software engineering approach that is based on a tool-supported, semi-automatic transformation of scenario-based, aspect-oriented requirements models into aspect-oriented business process design models. This approach is realized by a mapping from Aspect-oriented Use Case Maps (AoUCM) to ADORE business process models, allowing for the continued encapsulation of requirements-level concerns in design-level artifacts.

Inspired by *agile* methodologies, we consider that the design phase impacts the requirements. As a consequence, we defend an iterative process where both requirements and design artifacts interact together to build the final system. We illustrate this vision in FIG. 13.1.

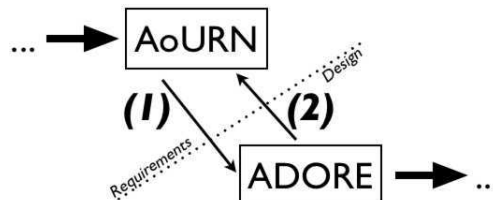


Figure 13.1: Overview of an AoURN ↔ ADORE process

The first step (1) in the process is to generate design artifacts based on the structural and behavioral information available in the requirements model. An automated transformation of

requirements models into design models reduces (or even eliminates) the amount of work done in the requirements phase that needs to be redone in the design phase while at the same time improving the consistency between the two layers. Changes at the requirements level are validated at this level, before being propagated to the design level. Moreover, due to the automatic, correctness-preserving transformation, we ensure that the generated design model conforms to these requirements. Hence, software engineers can take advantage of the analysis capabilities of requirements notations. The feedback from the analyses serves to improve both the requirements and the generated design models.

The reciprocal arrow (2) supports the transformation of design decisions into requirements ones, when useful. ADORE may identify *inconsistent* data-flows introduced by omissions in the requirements models (e.g., the requirements may assume the usage of data which are never defined explicitly in the system). This information is fed back to the requirements layer for iterative refinement. From the *separation of concerns* point of view, ADORE identifies interactions between several concerns around shared join points, which may not be described in the requirements model. Such interactions can then be solved at the requirements level, if necessary.

13.2.3 Firsts Experiments

We implement such a generative approach for AoUCM and ADORE models. For a given requirements model r , we define a model transformation τ to produce the associated design model $d = \tau(r)$. We based our first experiment on the PICWEB example. With the help of AoURN, each concern of the PICWEB application can be modeled from two points of view. AoURN goal models describe the reasons for including a concern in the application and the impact of the concern on high-level goals of all stakeholders. AoUCM, on the other hand, describes the scenarios of the concern as well as the actors and structural system entities that work together to realize the scenarios. We depict in FIG. 13.2 an AoURN representation of the PICWEB *getPictures* initial process.

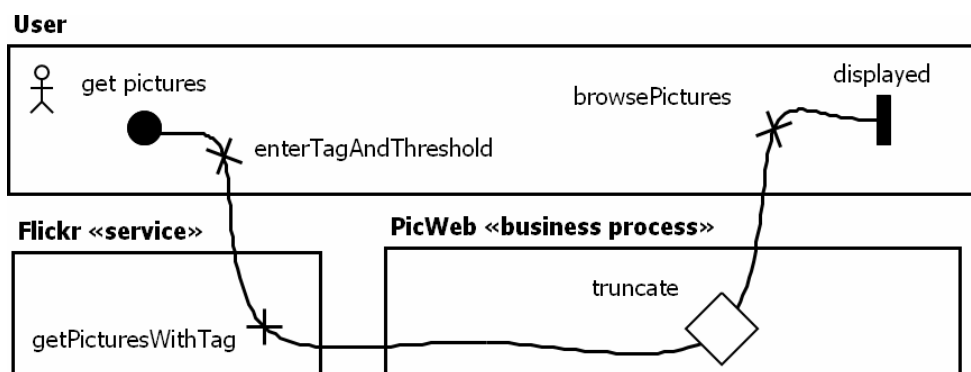


Figure 13.2: PICWEB *getPictures* process described in the AoURN notation

Based on the information described in the requirements, the transformation algorithm generates *all* the process skeletons associated with PICWEB concerns that are composed with each other as desired. The design artifacts expressed in the language daily used by the designer, and consistent according to the requirements. Designers only focus on their field of expertise and can identify problems in the system (such as omissions) without having to build a design model from scratch. The work effort is reduced, and the interventions are more accurate and precise. The requirements engineer, on the other hand, benefits from an updated requirements model which can be verified against the stakeholders' goal models and re-evaluated by requirements analysis techniques to ensure that the changes do not conflict with the intended results.

13.2.4 Summary

With this work, we sketched a model transformation from a requirement engineering tool to ADORE. This approach supports both requirement engineer and process designers while designing an application. In terms of software design, it emphasizes the need to automate the operationalization of requirements artifacts produced at the requirement steps. Moreover, we identified several situations where the business process design phase introduces changes in the requirements, which can be automatically capitalized with the associated transformation.

13.3 Downstream: Visualizing & Assessing Compositions

This work is the product of a collaboration with Bergel, University of Chile. In this work, we propose to link ADORE with a visualization engine (MONDRIAN) [Mosser et al., 2010a]. The definition of visualizations associated to ADORE artifacts supports designers by providing a holistic point of view on the compositions involved in the final system.

13.3.1 Context

Assuming that a complex system is difficult to understand by humans, separation of concerns mechanisms propose to reduce the complexity by defining several smaller artifacts instead of a single and large one. They identify and encapsulate parts of models that are relevant to a particular concern. These artifacts are then composed to produce the expected system. As a consequence, the intrinsic complexity of the system is shifted into the composition directives used to build it. When a system involves many processes, it is necessary to have a holistic point of view on features, compositions and business processes to grasp it.

When developing a large system (e.g., the CCCMS, CHAP. 11), designers handle a large set of initial orchestrations, and a large set of fragments to apply in these orchestrations. ADORE tackles the complexity of integrating fragments into orchestrations, at the activity level. Such a detailed and focused view of composition is not scalable to support the design of large system (non-functional concerns introduction in the CCCMS raises the number of unification to more than a thousand). Visualization techniques of large data set such as fish-eye [Sarkar and Brown, 1992] can tame the readability problems, but do not reduce the amount of details visualized in this representation. When designing a complete set of business processes using a compositional approach, the objectives of designers are to retrieve a holistic representation of the involved entities to understand easily what they are doing. Such a global visualization is needed at both design and analysis time:

- *Design Phase.* When building a complete system using a compositional approach, designer needs to understand at a coarse-grained level the interactions between the different composition entities they are manipulating. At this step, designers focus on the fragments in terms of *impact* (e.g., “the fragment throws a fault”) instead of their detailed behavior (e.g., “when a resource takes too much time to reach the crisis location, a timeout fault must be thrown”).
- *Analysis Phase.* After the composition directives execution, designers obtain a *composed* system. At this step, they need to identify critical points of the composed system, for example to design unit tests. This step focuses on the comparison between the *intrinsic complexity* of the original entities and the composed result.

13.3.2 Proposed Approach & Firsts Experimentations

We define three different visualizations of compositions. The common objective of these three visualizations is to provide *dashboards* to designers. Composition dashboards are graphical representations meant to help designers to (i) get a scalable and global overview of the compositions present in an orchestration-based application, (ii) identify abnormal compositions and (iii) facilitate the comprehension of a large composition by categorizing compounds. The idea of these

dashboards is to enable a better comparison of elements constituting a program structure and behavior. These visualization are obtained through a model transformation from the ADORE logical meta-model to an object-oriented one (defined in Smalltalk) used by MONDRIAN to define its graphical rendering. It is out of the scope of this section to fully describe these visualizations. The interested reader may refer to the associated publication. We focus here on a coarse grained visualization of fragments, used to assess the set of available fragments used to build the system. We use the CCCMS case study described in CHAP. 11.

The first visualization represents all the *fragments* (as square) involved in a system, focusing on their *impact*. We have identified several impact properties represented as *boxes*: (i) hooked variable modification, (ii) exception throw, (iii) fault handling, (iv) initial process execution inhibition and finally (v) restricted process inhibition. To illustrate this visualization, we instantiated it on the CCCMS example (Fig. 13.3). It represents the 24 fragments defined to answer to the different use-cases extensions realizations (based on the requirements).

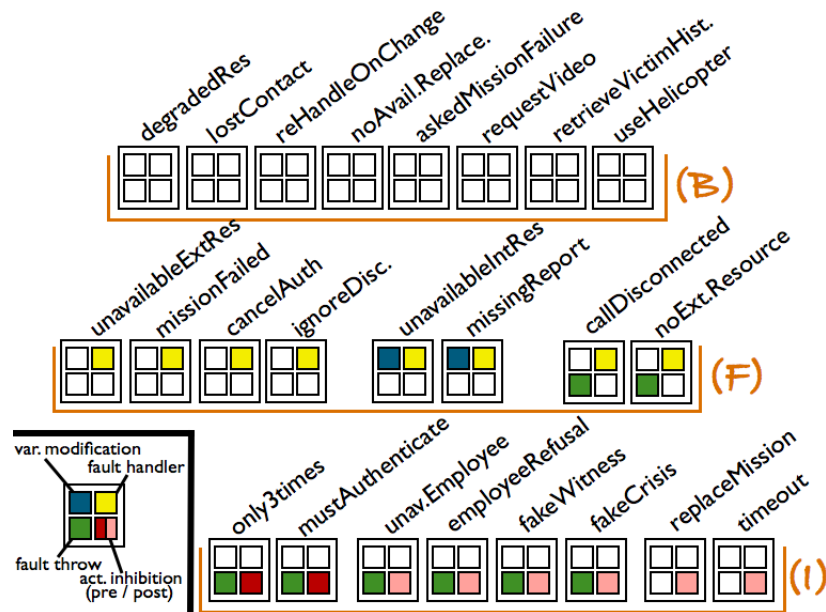


Figure 13.3: Fragments dashboard instantiated on the CCCMS example.

Interpretation. Based on the graphical representation obtained in this view, we have identified 7 different fragment categories, grouped into 3 main families: business extensions (\mathcal{B}), fault handling (\mathcal{F}) and control-flow inhibition (\mathcal{I}).

- Business extensions (\mathcal{B}): The *white* fragments only enrich existing process with new additional behavior. They do not modify the initial logic of the business process, and only add new features to enrich it.
- Fault handler (\mathcal{F}): *Yellow* boxes represent *fault handling* property. There are several ways to deal with a fault when it occurs in a process: (i) doing a re-throw (*green* property) to customize the fault, (ii) bypassing the fault (by modifying data to handle the problem, *blue* property) locally and (iii) handling the fault by using a business-driven reaction (no other property).
- Control-flow inhibition (\mathcal{I}): The *red* property represents the inhibition of an activity and its followers in a business process. The *pink* property is a restriction of the *red* one, since the fragment only inhibits the followers of an activity. Correlated with the fault thrower property (*green* color), we can identify *precondition* (activity inhibition & throw) and *post-condition* (followers inhibition & throw) checker. The *pink-only* fragments represent “dangerous” fragments, which inhibit several activities using a non-standard behavior. A deep

understanding of the business–domain is necessary to grasp their behavior properly. The `timeout` fragment is a typical example of this kind of fragments: instead of “simply” throwing an exception when the system detects that a resource takes too much time to reach the crisis location, the CCCMS business model asks to stop whatever the system was doing and urgently inform a human coordinator able to decide what to do to counterbalance the situation.

13.3.3 Summary

The described view helps designers to perform a coarse–grained identification of their critical fragments, and assess the others. Based on these preliminary results, we submit a project to the Chile Research Agency (ECO-Sud program) to facilitate further work. We plan to investigate other representations associated to compositions, based on processes metric indicators. Moreover, others indicators can be used to assess compositions, such as data–flow treatment, security introduction, performances costs.

13.4 Sketched Perspectives

The two previous perspectives were initiated with external collaborations. These collaborations still need to be enriched to enrich the preliminary results identified in the associated articles. The following perspectives are in a more prospective state, and describes directions we think interesting to follow to enhance ADORE.

13.4.1 Users Experiments

The implementation of ADORE was realized to support the approach defended in this dissertation. As a consequence, it is a very young platform. Even if we successfully use the tool on five case study (only two were described here for concision reasons), the tool was never used by others than our team. To address this issue, we plan to set up a *blind* experiment where the same application is modeled by process designers with (i) usual tools and (ii) ADORE.

13.4.2 ADORE Expressiveness & Tooling

In this document, we describe the ADORE framework, and how it is used to handle several case studies. However, the framework suffers from several limitations when talking about its expressiveness or tooling.

Code Generation: From ADORE to BPEL. ADORE is a model–driven framework. It works on design models, and allows designers to compose business processes independently of their implementation. We sketch in [Blay-Fornarino et al., 2009] how ADORE models can be transformed into executable BPEL processes, using the KERMETA¹ meta–language to support the transformation. However, more studies should be conducted in this area to handle the transformation in the general case. Based on this code generation transformation, a complete development process can be realized through ADORE (from requirements to code).

Representation of Data Structures. The ADORE framework focuses on behavior representation. At the model level, several works address the structural composition challenge (e.g., KOMPOSE [Fleurey et al., 2007]). Based on a code–driven experimentation made by fourth–year engineer students, we identify in the context of the CCCMS several incomplete structural models which can be derived from ADORE artifacts. To define a complete model–driven framework to support composition of business process (both structural and behavioral), we are considering to identify and then maintain a link between ADORE and KOMPOSE models. This link will ensure the consistency of the final system, binding the two framework into a SOA dedicated one.

¹<http://www.kermeta.org>

Iteration Policies Composition & Parallelism. As far as we are, we do not address in ADORE the composition of iteration policies. State of the art research on grid-computing and workflow enactment [Glatard, 2007] focuses on such problematic, at the workflow level. By enhancing the expressiveness of ADORE, one may define several policies on the same activities, and then express how the data are composed. These information can be used to define a more clever DATAFLOW algorithm, able to melt policies between each others.

13.4.3 Actions Sequences & Optimizations

The ADORE framework defines “algorithms” as action generators. These actions are then executed sequentially by the execution engine. We identify two directions to follow to optimize this methodology: (i) sequence optimization and (ii) multi-threaded execution.

Sequence optimization. Even on simple example, we show that the generated sequences of actions are not *minimal*. An a-priori analysis of the sequences may effectively reduce the length of these artifacts. Another direction is the optimization of sequences based on the action semantics: if an element e is going to be deleted with the execution of the action a_i , one may consider to not execute actions which modifies e before its deletion (at the end, it will be deleted, so why should we waste time on this element?). These optimizations are not that easy [Blanc et al., 2008], but may lead to an approximation of minimal sequences.

Execution optimization. Nowadays, multi-core architecture are usual, even on personal computer. The use of multiple threads in an application can *speedup* its execution time. In the context of ADORE, the action sequence to be executed can be analyzed to identify disjoint actions (that is, activity which does not work on the same artifacts). Consequently, a thread pool can be used to introduce parallelism inside the execution engine. Following the same idea, algorithms invocation are intrinsically independent (i.e., “*embarrassingly parallel*”), and may be executed in parallel (even automatically [Dutra et al., 2004]). We do not provide an extensive benchmark, but our preliminary tests on toy examples show a real benefit (approximatively two times faster on a dual-core processor) to multi-thread the composition engine.

13.4.4 Knowledge Representation & Semantic Web

The knowledge basis used by ADORE is very restrictive. It is only a technical solution to a more general problem. We sketched in the JSEDUITE case study how the knowledge basis expressiveness can be enhanced to automatically infer interference resolution rules. However, this solution is not optimal as dedicated to a given application. We also show (see SEC. 12.4) that a semantic point of view [Gasevic et al., 2006] on ADORE artifacts should supports designers through the automation of several reasoning.

Globally, the knowledge basis used by ADORE can be seen as an “expert system”, and the interference resolution rules as “business rules”. Coupled with a semantic web engine to express the relationships between services and exchanged data, an important part of the interference resolution may be automated. This idea is inspired by the one described in [Moreau et al., 2009], where the authors automatically repair data-flow in BPEL processes, based on semantic annotations.

13.4.5 Dynamic Weaving

The ADORE framework is a static one. It supports the static definition of behavioral models, and their compositions. But the emerging “Models at Runtime” paradigm [Garlan and Schmerl, 2004] emphasizes the need to handle models compositions at runtime, following a dynamic approach. Moreover, context-aware application [Parra et al., 2009] needs to be reconfigured at runtime, which induces a dynamic modification of their behavior when encountering a new context of execution. For all these reasons, we found very important to now think about ADORE as a dynamic composition engine, identifying what should be enhanced to support such an approach.

«It is good to have an end to journey toward; but it is the journey that matters, in the end.»

Ursula K. Le Guin

In this document, we present the core of our research dealing with composition mechanisms dedicated to Service-Oriented Architectures. To digest the contribution of this manuscript, we take each part of the document, and then summarize its associated contribution.

State of the Art. From the literature, we particularly study two fields: (i) compositions of services and (ii) compositions of models. The first one provides solutions to support designers when they compose several services into more complex ones. It defines languages, norms and framework which efficiently perform the assembly. From these mechanisms, we restrict the scope of this study to business processes and workflow design. In front of the multiplicity of norms and languages, we decide to follow a model-driven approach, being independent of a given technical platform. The second field tackles the intrinsic limit of the first approaches. It supports (under the name of “Separation of Concerns”) the design of large systems through the expression of several small artifacts, and let the hard task of combining these incomplete models into the final one to automatic engine. However, there is no method dedicated to the design of business processes following the separation of concerns paradigm, at the model level. Filling this gap is the objective of this thesis. We summarized in TAB. 14.1 the different answers provided by ADORE regarding the properties P_i and Π_i identified in this study of the literature.

The ADORE Kernel. Based on our study of the literature, we propose a meta-model named ADORE able to reify key concepts associated to business process. We propose a graph-based meta-model, where process activities are represented as nodes, and relations between activities (e.g., wait or guards) as edges. The meta-model handles concepts dedicated to support a compositional approach of business process design, with the definition of *fragment* of processes. These fragments are incomplete by nature, and aim to be integrated into others processes. We associated to this meta-model a graphical syntax and a formal execution semantic which underly the remaining of the manuscript. Finally, we propose an action-based language used to interact with existing models conform to the ADORE meta-model.

Composition Algorithms. The ADORE kernel allows us to define several algorithms to support the design of large business processes. We identified four algorithms, described as the following:

- The WEAVE algorithm aims to integrate fragments into other processes. An interesting characteristic of this algorithm is its order-independence: the obtained result does not depend on the execution order of the weave directives expressed by the designers. As a consequence, it is intrinsically different (but hopefully complementary) to sequential approaches such as features or aspects.
- The MERGE algorithm handles the shared join point issue, triggered when several concerns must be integrated in the base process at the same location. Instead of being ordered, we propose to merge the involved concerns, keeping them explicitly unordered. Based on this algorithm, we propose a methodological approach which supports designers while building complex processes.
- The INLINE algorithm was defined for performance purpose. Based on experimental results, we identify that this well-known compilation technique makes sense in the context of business process. As a consequence, we give to process designers an automatic capability for inlining processes into other ones.
- Finally, the DATAFLOW algorithm is inspired by the grid-computing community. With this algorithm, we consider that process designers are not supposed to be computer scientist, and then should not be annoyed with technical details such as loops. We define this algorithm to automatically introduce an iteration over a data set. With this method, one can describe a process which works on a scalar, and then lets the algorithm tackle the complexity of identifying the activities involved in the iteration, and finally creates the associated artifacts in the original model.

Applications. According to Richard Feynman, *“For a successful technology, reality must take precedence over public relations, for nature cannot be fooled.”*. We illustrate the usage of ADORE with two large case studies. The first one was the definition of a Car Crash Crisis Management System (CCCMS), according to a requirements document co-published by the University of McGill and the University of Luxembourg. This study shows how ADORE is used to support the design of a complex system, defined by someone external. The second case study deals with information broadcasting inside academic institutions. This application (JSEDUITE) is developed by our research team (the first release was published in 2005) and can be considered as a “medium-to-large” system (about 70,000 lines of code). We accurately use ADORE to reverse engineer the legacy system (deployed into several institutions), and tackle methodological issues raised while developing the system at the code level.

Finally, we expose as perspectives several research topics identified as perspectives of ADORE. Some of them are based on ongoing collaborations with other universities, but most of them are open fields which may lead to very exciting research dealing with composition and business processes.

“That’s all folks!”
— The Looney Tunes

	SOA Property	ADORE Answer
P_1	Business Expressiveness	The ADORE meta-model (see CHAP. 4) defines <i>activities</i> and <i>relations</i> between activities, that is, the essence of business process modeling. Fragments are described using the <i>same</i> formalism. The composition directives are expressed in terms of ADORE activities, in business terms.
P_2	One cannot ignore reality	The ADORE meta-model can be seen as the simplification a highly adopted standard (BPEL). We also use activity-diagrams like notation so support the design of processes. We assess the meta-model with two large case studies (see PART IV)
P_3	Concern Reification	Concerns are reified as <i>fragments</i> , using the ADORE formalism (see CHAP. 4)
P_4	Behavioral Composition	ADORE composition mechanisms (see PART III) support the enhancement of preexisting business processes, in an order-independent way.
P_5	Activity Parallelism	The meta-model relies on a partial order between activities and consequently allow naturally such an expressiveness. The compositions algorithms never introduce an order if not needed (see PART III), and then conserve the preexisting parallelism
P_6	Syntax Independence	Even if ADORE is defined as a simplification of the BPEL, its intrinsic concepts are not specific to the language.

	SOC Property	ADORE Answer
Π_1	Concurrent experts support	The MERGE and WEAVE algorithms support an order-independent composition process (see CHAP. 8). Expert can work concurrently; the way they order their directive does not matter at the end.
Π_2	Multiple comp. algorithms	The action framework (see CHAP. 6) support the definition of composition algorithms in a simple way. We defines four different composition algorithms in PART III.
Π_3	Comp. as first class entities	Composition directives are reified as first-class entities, and the <i>algorithm scheduler</i> (see CHAP. 7) reason on these entities to support the order-independence.
Π_4	Conflict detection	Conflict detection mechanisms (defined as boolean predicate satisfaction) are used in the approach to support the assessment of composed elements (see CHAP. 7).

Table 14.1: Summary of the contribution, according to PART I properties

Appendices



Implementation of ADORE

«Theory guides. Experiment decides»

Contents

A.1 Tool Support	195
A.2 Composition Algorithms Invocation	197
A.3 Pointcuts	197
A.4 Fact Model & Associated Action Language	198
A.5 Execution Example	199
A.6 Execution Semantics	200

In this chapter, we briefly describe the current implementation of the ADORE platform. This description is based on a demonstration designed for the AOSD'10 conference. The complete implementation of ADORE is available under a LGPL license, on the associated public repository¹. This document is based on the revision 205 of ADORE.

A.1 Tool Support

ADORE user interface is implemented as an EMACS major mode, as shown in Fig. A.1. This mode hides in a user-friendly way the set of shell scripts used to interact with the underlying engine. The concrete ADORE engine is implemented as a set of logical predicates, using the PROLOG language. Interference detection rules are also implemented as PROLOG predicates.

A dedicated compiler (defined using ANTLR²) implements an automatic transformation between ADORE concrete syntax and the associated PROLOG facts used internally by the engine (Fig. A.2). As visualizing processes is important in design phase, ADORE provides a transformation from its internal facts model to a GRAPHVIZ³ code which can then be compiled into a graphic file (e.g., PNG, EPS). It produces as a result a graphical representation of ADORE models, as depicted in all the figures showing ADORE models in this document.

Raw data (e.g., number of activities, relations, process width) can be extracted as a XML document. This document can then be processed (manually or using technology like XSLT) to produce *before/after* graphics and benchmark the approach, as shown in the CCCMS validation CHAP. 11.

¹<http://code.google.com/p/adore>

²<http://www.antlr.org/>

³<http://www.graphviz.org/>

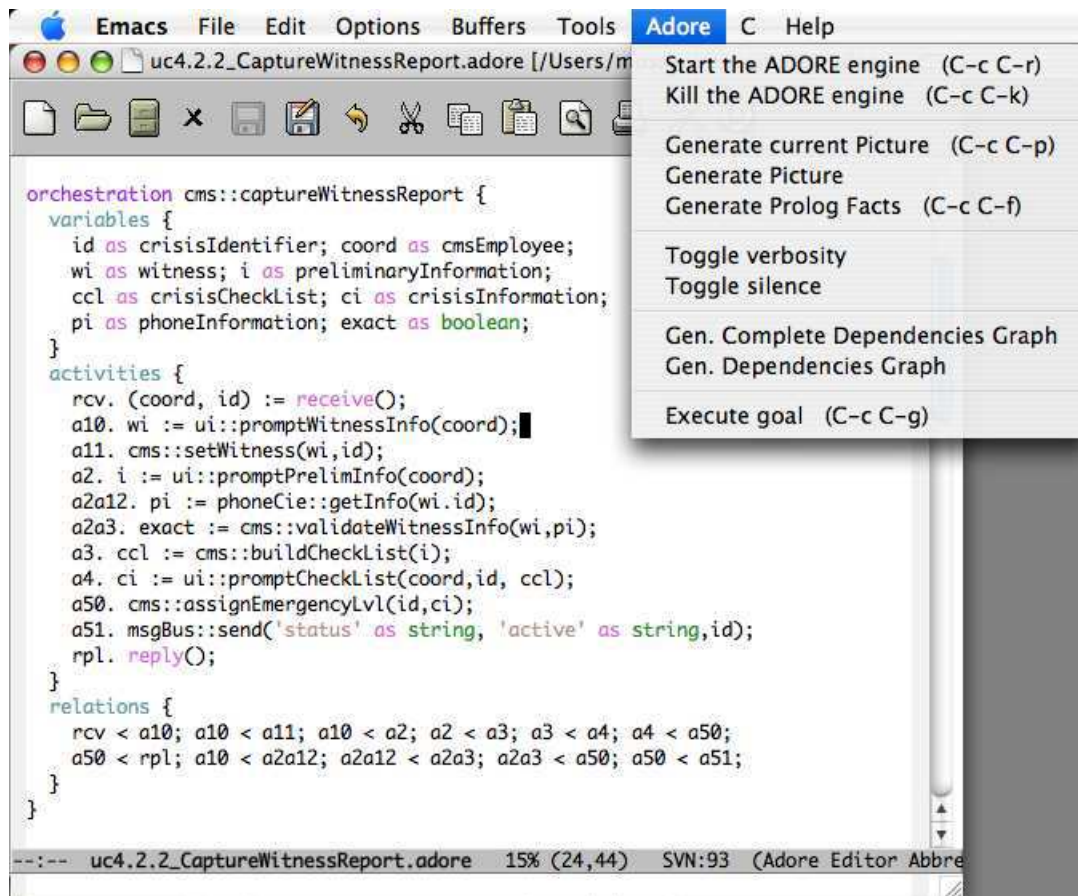


Figure A.1: ADORE editor, as an EMACS major mode

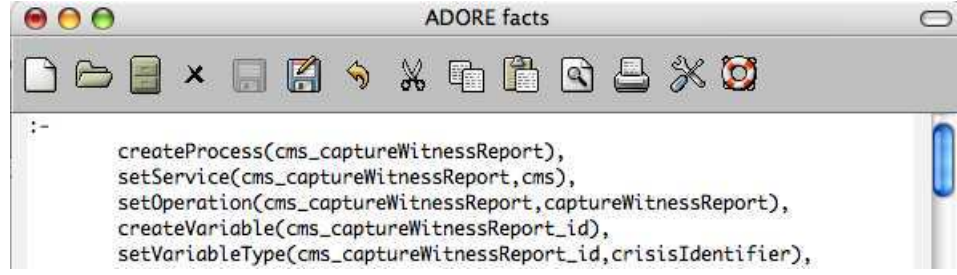


Figure A.2: PROLOG facts, generated by the ADORE compiler

A.2 Composition Algorithms Invocation

The ADORE surface language allows modelers to define composition units. These compositions are compiled as PROLOG facts and analyzed by the engine. When the modeler asks ADORE to run the compositions, it determines for a given composition the different algorithms (e.g., fragment weave, behavioral merge) that need to be triggered and execute them.

Based on the CCCMS case study, we consider as an example a subpart of the `captureWitnessReport` composition. We focus here on the a_4 activity, where the modeler asks to weave two different fragments (`ignoreDisconnection` & `fakeCrisisDetected`):

```
composition cms::captureWitnessReport {
  apply ignoreDisconnection => a4;
  apply fakeCrisisDetected  => a4;
}
```

To perform such a composition, the logical engine schedules the algorithms invocations, represented in LISTING A.1. It starts by *cloning* the two initial fragments into temporary entities (lines 1 & 2). The identification of a shared join point triggers the *merge* of the two involved fragments into a new one (line 3). Finally, this merged fragment is *woven* on the initial process (line 4), and a graph simplification algorithm is triggered (line 5) to make the result more understandable by humans (retracting useless relations).

```
1 doClone(ignoreDisconnection, tmp_1),
2 doClone(fakeCrisisDetected, tmp_2),
3 doMerge([tmp_1,tmp_2], merged_1),
4 doWeave([weave(merged_1, [cms_captureWitnessReport_a4])]),
5 doProcessSimplification(cms_captureWitnessReport),
```

Listing A.1: Internal algorithms usage (automatically computed)

A.3 Pointcuts

ADORE does not provide any surface mechanisms to deal with pointcuts expressiveness. One can start the engine in interactive mode and directly use PROLOG to query the model using logical unification. Losing the surface language implies that the modeler knows the PROLOG programming language. But the immediate benefits are the unbounded possibilities of pointcut description offered by PROLOG.

We consider as an example the pointcut associated to the `logUpdate` fragment. This fragment must be applied on service invocations which use a variable of type `cmsEmployee` or `worker` as input. Based on the ADORE formal model, such a pointcut can be expressed as the following logical rule ($a \in acts(p)$):

$$Kind(a) = invoke \wedge \exists v \in InputVars(a), Type(v) = (cmsEmployee \vee worker)$$

Using PROLOG, it is very easy to implement such a rule, as shown in LISTING A.2. We provide on the website a PROLOG *toolbox*⁴ to automate recurring patterns identification.

```

1 cut4logUpdate(Acts) :-
2   findall([A], (findVarUsageByType(cmsEmployee,in,A), hasForKind(A,invoke)), CmsEmployees),
3   findall([A], (findVarUsageByType(worker,in,A), hasForKind(A,invoke)), Workers),
4   merge(CmsEmployees, Workers, Raws), sort(Raws, Acts).

```

Listing A.2: PROLOG code associated to the logUpdate non-functional concern

We provide in ADORE a meta-predicate (LISTING A.3) used to automate the pointcut mechanism. This predicate calls the user-defined pointcut predicates and then automatically builds the associated *weave* directives.

```

1 build(FragmentName, PointcutPredicate, Directives) :-
2   call(PointcutPredicate, List), findall(weave(FragmentName,E), member(E,List), Directives).

```

Listing A.3: ADORE Meta-predicate used to automate pointcut matching

One can then define a pointcut as a unary predicate and asks ADORE to build the composition directives associated to it, as shown in (LISTING A.4).

```

1 ?- build(logUpdate, cut4logUpdate, L), length(L, Count).
2 L = [weave(logUpdate, [a13]), weave(logUpdate, [a20]),|...],
3 Count = 72.

```

Listing A.4: Computing composition directives associated to logUpdate

A.4 Fact Model & Associated Action Language

We implement the ADORE logical model as a set of elementary facts. As a consequence, the implementation is more fine-grained than the formal model. It allows us to be more efficient at the implementation level while handling ADORE models. We represent in LST A.5 the dynamic PROLOG fact model used in ADORE, and the associated actions used to interact with it.

```

1 %% Business Processes (Orchestration / Fragment)
2 :- dynamic process/1.           %% createProcess(P)
3 :- dynamic isFragment/1.        %% setAsFragment(P)
4 :- dynamic hasForParameter/2.   %% setAsFragmentParameter(P,I)
5 :- dynamic hasForSrvName/2.     %% setService(P,I)
6 :- dynamic hasForOpName/2.      %% setOperation(P,I)
7 :- dynamic hasForColor/2.       %% defColor(P,I)
8
9 %% Activities
10 :- dynamic activity/1.          %% createActivity(A)
11 :- dynamic hasForKind/2.        %% setActivityKind(A,K)
12 :- dynamic isContainedBy/2.     %% setContainment(A,P)
13 :- dynamic hasForService/2.    %% setInvokedService(A,I)
14 :- dynamic hasForOperation/2.  %% setInvokedOperation(A,I)
15 :- dynamic usesAsBinding/3.     %% setMessageBinding(A,I,V)
16 :- dynamic hasForFunction/2.   %% setFunction(A,I)
17
18 %% Variables
19 :- dynamic variable/1.          %% createVariable(V)
20 :- dynamic hasForType/2.        %% setVariableType(V,T)
21 :- dynamic hasForInitValue/2.   %% setInitValue(V,S)
22 :- dynamic isConstant/1.       %% setConstancy(V).
23 :- dynamic isSet/1.            %% flagAsSet(V).

```

⁴such as the findVarUsageByType predicate which identify activities that use a specific type in their associated variables.

```

24 :- dynamic usesAsInput/2.      %% addAsInput(V,A)
25 :- dynamic usesAsOutput/2.    %% addAsOutput(V,A)
26 :- dynamic fieldAccess/3.     %% createFieldAccess(I,V,L).
27
28 %% Relations
29 :- dynamic waitFor/2.         %% defWaitFor(A,A)
30 :- dynamic weakWait/2.        %% defWeakWait(A,A)
31 :- dynamic isGuardedBy/4.     %% defGuard(A,A,V,true|false)
32 :- dynamic onFailure/3.       %% defOnFail(A,A,S|'')
33
34 %% Policies
35 :- dynamic policy/3.          %% defPolicy(I,Formula,Formula)
36 :- dynamic iteratesOver/2.    %% setIteration(A,Policy)
37
38 %% Composition directives
39 :- dynamic context/1.         %% defCompositionContext(I)
40 :- dynamic contextTarget/2.   %% setCompositionTarget(I,P)
41 :- dynamic contextOutput/2.   %% setContextOutput(I,I)
42 :- dynamic activityBlock/3.   %% defActivityBlock(I,I,L)
43 :- dynamic applyFragment/4.   %% defApply(I,P,I,P)
44 :- dynamic applyParameter/3.  %% setApplyParam(I,I,S)
45 :- dynamic setDirective/2.    %% defSetify(I,V)
46
47 %% Knowledge
48 :- dynamic adoreEquivalence/2.
49 :- dynamic variableBinding/2.

```

Listing A.5: Implementation of the ADORE logical model as PROLOG facts

A.5 Execution Example

The ADORE engine defines a mechanism to keep trace of the actions generated by the algorithm. We describe here the trace generated while performing the compositions depicted in LST A.1.

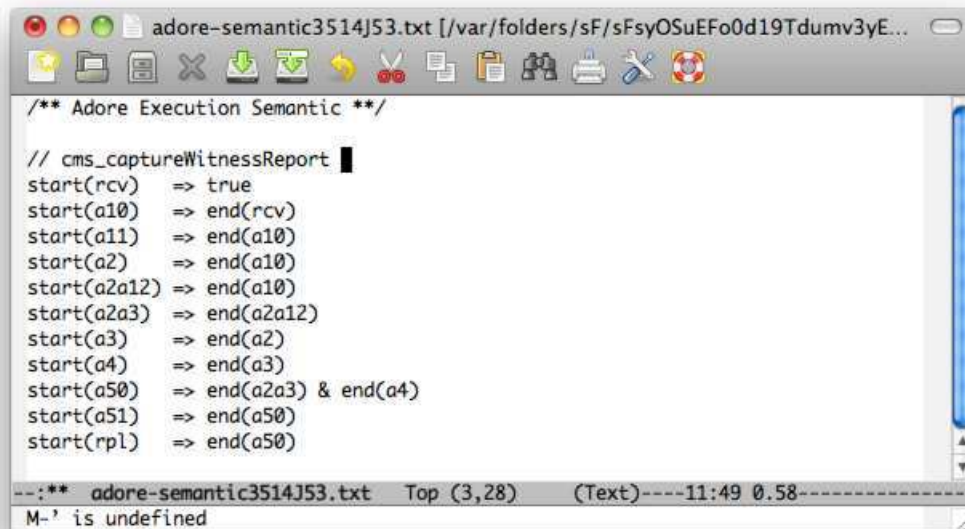
```

% INFO: Running doClone(ignoreDisconnection,uc22_id_1)
% INFO:   Computing action set
% 3,016 inferences, 0.000 CPU in 0.001 seconds (0% CPU, Infinite Lips)
% INFO:   => Result: 14 actions
% INFO:   Executing action set
% 490 inferences, 0.000 CPU in 0.000 seconds (0% CPU, Infinite Lips)
% INFO: doClone(ignoreDisconnection,uc22_id_1) ended with success!
% INFO: Running doClone(fakeCrisisDetected,uc22_fCD_2)
% INFO:   Computing action set
% 3,525 inferences, 0.010 CPU in 0.001 seconds (772% CPU, 352500 Lips)
% INFO:   => Result: 36 actions
% INFO:   Executing action set
% 1,300 inferences, 0.000 CPU in 0.000 seconds (0% CPU, Infinite Lips)
% INFO: doClone(fakeCrisisDetected,uc22_fCD_2) ended with success!
% INFO: Running doMerge([uc22_id_1, uc22_fCD_2],uc22_id_fCD_1)
% INFO:   Computing action set
% 1,835 inferences, 0.000 CPU in 0.001 seconds (0% CPU, Infinite Lips)
% INFO:   => Result: 17 actions
% INFO:   Executing action set
% 4,984 inferences, 0.000 CPU in 0.001 seconds (0% CPU, Infinite Lips)
% INFO: doMerge([uc22_id_1, uc22_fCD_2],uc22_id_fCD_1) ended with success!
% INFO: Running doWeave(...)
% INFO:   Computing action set
% INFO:   weave(uc22_id_fCD_1,[cms_captureWitnessReport_a4])
% 5,994 inferences, 0.000 CPU in 0.002 seconds (0% CPU, Infinite Lips)
% INFO:   => Result: 18 actions
% INFO:   Executing action set
% 6,544 inferences, 0.000 CPU in 0.001 seconds (0% CPU, Infinite Lips)
% INFO: doWeave(...) ended with success!
% INFO: doProcessSimplification(cms_captureWitnessReport) ...
% 258,933 inferences, 0.040 CPU in 0.036 seconds (113% CPU, 6473325 Lips)
% 14,885 inferences, 0.000 CPU in 0.002 seconds (0% CPU, Infinite Lips)
% INFO: ... done (58 actions).

```

A.6 Execution Semantics

The execution semantics computation described in SEC. 5.4 is implemented in the engine. Based on a process model expressed using the ADORE language, one can run the algorithm and then obtain a textual description of the boolean formulas associated to each activity, as depicted in FIG. A.3.



The screenshot shows a text editor window titled 'adore-semantic3514J53.txt' with a standard toolbar. The text content is as follows:

```

/** Adore Execution Semantic */

// cms_captureWitnessReport
start(rcv) => true
start(a10) => end(rcv)
start(a11) => end(a10)
start(a2) => end(a10)
start(a2a12) => end(a10)
start(a2a3) => end(a2a12)
start(a3) => end(a2)
start(a4) => end(a3)
start(a50) => end(a2a3) & end(a4)
start(a51) => end(a50)
start(rpl) => end(a50)

```

The status bar at the bottom indicates the file path, line 3 of 28 characters, text mode, and a timestamp of 11:49 0.58. A message 'M-' is undefined' is also visible.

Figure A.3: Execution semantics formulas, automatically computed by the engine

Algorithms Implementation

«For a successful technology, reality must take precedence over public relations, for nature cannot be fooled.»

Richard Feynman

Contents

B.1 Algorithm “Framework” & Architecture	201
B.2 Complete Example: Process Simplification	202
B.3 Chaining Algorithms	203

B.1 Algorithm “Framework” & Architecture

We implement composition algorithms as plain PROLOG source files. In addition to the *four* final algorithms presented in this document, we publish in the source repository several other prototypes:

<http://adore.googlecode.com/svn/trunk/prolog/algos/>

Algorithms definition relies on the *module* system defined by the SWI-PL implementation of PROLOG. Each algorithm is described in a given module, and then cannot interfere with others (e.g., name conflicts, multiply defined predicates).

ADORE algorithms follow the same implementation pattern. An algorithm *X* defines a predicate **buildActions**, which generates ADORE actions to be executed on the current universe. A predicate **doX** is defined as a shortcut to (i) compute the action set and (ii) execute it on the current universe. This predicate is exported outside the module, and is consequently available for the user. The remaining of the file contains algorithm–dedicated predicates.

We consider here the WEAVE algorithm to illustrate this architecture. The first line declares this code as a module for the SWI-PL module system, and publishes the **doWeave** predicate. Lines 3 and 4 implement the **doWeave** predicate. It computes the actions associated to the received set of **Directives**, and then executes it. Finally, lines 6, 7, 8 are dedicated to the generation of the actions associated to the WEAVE algorithm: it starts by creating a new context associated to this invocation, and then retrieves the actions associated to each directives received as input.

```

1 :- module(weave,[doWeave/1]).
2
3 doWeave(Directives) :-
4     weave:buildActions(Directives, Actions), executeActionSet(Actions).
5
6 buildActions(Directives, Actions) :-
7     declareContext(weave(Directives),Ctx),
8     findall(A, weave:applyDir(Ctx, Directives,A), Raw), flatten([Raw],Actions).
```

Listing B.1: Illustration of algorithms architecture (WEAVE)

B.2 Complete Example: Process Simplification

We motivate in the previous section the need for a simplification algorithm, able to delete redundant relations in a given process. This simplification is defined according to the following principles:

- *waitFor Absorption.* The idea of this step is to identify relation which are implied by other ones. For example, defining between x and y a *guard* **and** a *waitFor* is redundant: the *guard* implies the *waitFor* semantic.
- *weakWait Simplification.* According to the execution semantic associated to ADORE, defining a **single** relation of king *weakWait* between two activities x and y is equivalent to the definition of a *waitFor*.
- *Transitive paths destruction.* Considering a relation $x < y$, it can be deleted if an equivalent transitive path $x < z \xrightarrow{\sim} y$ exists in the process.

For illustration purpose, we use three different implementations mechanisms for these principles, described in LST B.2. Lines 1 \rightarrow 10 declare the algorithm, and implement its public interface. The two first occurrences of the **simplify** predicate (lines 11 \rightarrow 16) generate the retract of a *waitFor* relation, according to the *waitFor Absorption* principles. We base the generation of this delete action on the PROLOG implementation of the ADORE meta-model. The final occurrence of the **simplify** predicate (lines 17.18) generates two actions, according to the *weakWait Simplification*. When a single *weakWait* is detected, the predicate retracts it and defines the equivalent *waitFor*. This predicate uses the meta-call predicate **findall** to retrieve all the solution existing to a given goal. Algorithm can use the complete expressiveness power of PROLOG, and then express in a few lines of code very complex goals. Finally, we implement the last principle as a new composite action. This action **delTransitivePath** is declared as a **macroAction** in the engine (line 21), which will perform its refinements into elementary actions before trying to execute them. Such a syntactic shortcut allows users to dynamically enhance the action language for their own needs.

```

1 :- module( processSimplification, [doProcessSimplification/1, delTransitivePaths/2]).
2
3 doProcessSimplification(Process) :-
4     processSimplification:buildActions(Process,Actions), executeActionSet(Actions).
5
6 buildActions(Process, Actions) :-
7     process:exists(Process), process:getActivities(Process,Acts),
8     findall(A,processSimplification:simplify(Acts,A),SplActs),
9     flatten([ SplActs, delTransitivePaths(Process)], Actions).
10
11 simplify(Activities,myRetract(waitFor(X,Y))) :-
12     member(X, Activities), member(Y, Activities), waitFor(X,Y),
13     (isGuardedBy(X,Y,_,_) | weakWait(X,Y)).
14 simplify(Activities,myRetract(waitFor(X,Z))) :-
15     member(X, Activities), member(Y, Activities), member(Z, Activities),
16     isGuardedBy(X,Y,_,_), waitFor(X,Z), relations:existsControlPath(Z,Y).
17 simplify(Activities,[myRetract(weakWait(X,Y)),defWaitFor(X,Y)]) :-
18     member(X, Activities), findall(P,weakWait(X,P),[Y]).
19
20
21 :- assert(user:isMacroAction(delTransitivePaths,2)).
22 delTransitivePaths(Process,Actions) :-
23     findall(A, processSimplification:delATransitivePath(Process,A),Raw),
24     flatten(Raw,Actions), cleanup.
25
26 delATransitivePath(Process,Action) :-
27     process:getActivities(Process,Activities), member(X, Activities),
28     member(Y, Activities), waitFor(Y,X), relations:getControlPath(X,Y,L), \+ L = [X,Y],
29     getLazyGuards(X,GuardsX), getLazyGuards(Y,GuardsY), GuardsX = GuardsY,
30     relations:delAPath(X,Y,Action).

```

Listing B.2: Illustration ADORE algorithm implementation: SIMPLIFY

B.3 Chaining Algorithms

At the application level, we identify situations where a generic fragment can be defined and then instantiate to fit specific purpose (see SEC. 8.5, SEC. 12.3). For example, a fragment is defined to silently ignore a fault \mathbf{f} , and instantiated by the designer to catch an **overflow** fault in a given process. This algorithm is seen as an extension of the CLONE one: the internal content is cloned, and then the expected “generic” elements are substituted by new ones.

We illustrate this capability of ADORE in LST B.3, which describes the **buildActions** predicate of the INSTANTIATE algorithm. This predicate starts by creating a context (line 2), shared by this invocation and the underlying one. Then, it asks the CLONE algorithm to compute its associated actions (line 3). These actions are enriched by the substitutions identified to perform the instantiation (line 4), and finally returned to the user.

```

1 buildActions(Orig,Target,Output,Params,Dirs) :-
2     declareContext(instantiate(Orig,Target,Output),Ctx),
3     clone:cloneProcess(Ctx,Orig,Output,CloneActions),
4     performInstantiations(Ctx,Params,InstActions),
5     flatten([CloneActions,InstActions],Dirs).
```

Listing B.3: Invoking an algorithm from another one: INSTANTIATE

Simulating Aspect-Ordering with ADORE

«In order to be irreplaceable one must always be different.»

Coco Chanel

Contents

C.1 Initial Artifacts	205
C.2 Default Behavior Defined by ADORE	207
C.3 Simulating “Layered” Weaving	207
C.4 Introducing Aspect Ordering	207
C.5 ASPECTJ Implementation	211

Aspect-Oriented mechanisms usually define ordering mechanism to handle aspect woven at shared join points. We propose in ADORE a *merge*-driven mechanism. However, even if ordering techniques are not explicitly reified in ADORE, one can chain the invocations of the composition algorithms to simulate it. The goal of this chapter is to describe how the ASPECTJ weave semantic can be simulated in ADORE.

C.1 Initial Artifacts

We consider in this chapter a *toy-example*, depicted in FIG. C.1. We consider its *do* activity as the target of all the fragments defined in this chapter.

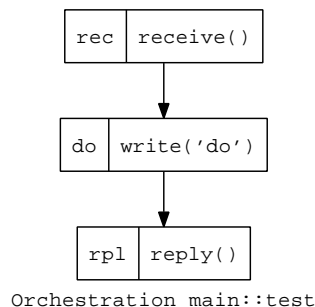


Figure C.1: Initial process used to illustrate aspect ordering in ADORE

ASPECTJ defines three kinds of *advice*s: **before**, **after** and **around**. ADORE does not provide such a difference at the model level, and considers all fragments as **around** advices. As a

consequence, we use the following conventions to simulate this mechanisms: (i) **before** fragments cannot define activities scheduled after the *hook* and (ii) **after** advices cannot define activities schedule before the *hook*. We represent in FIG. C.2 the fragments used in the context of this experiment.

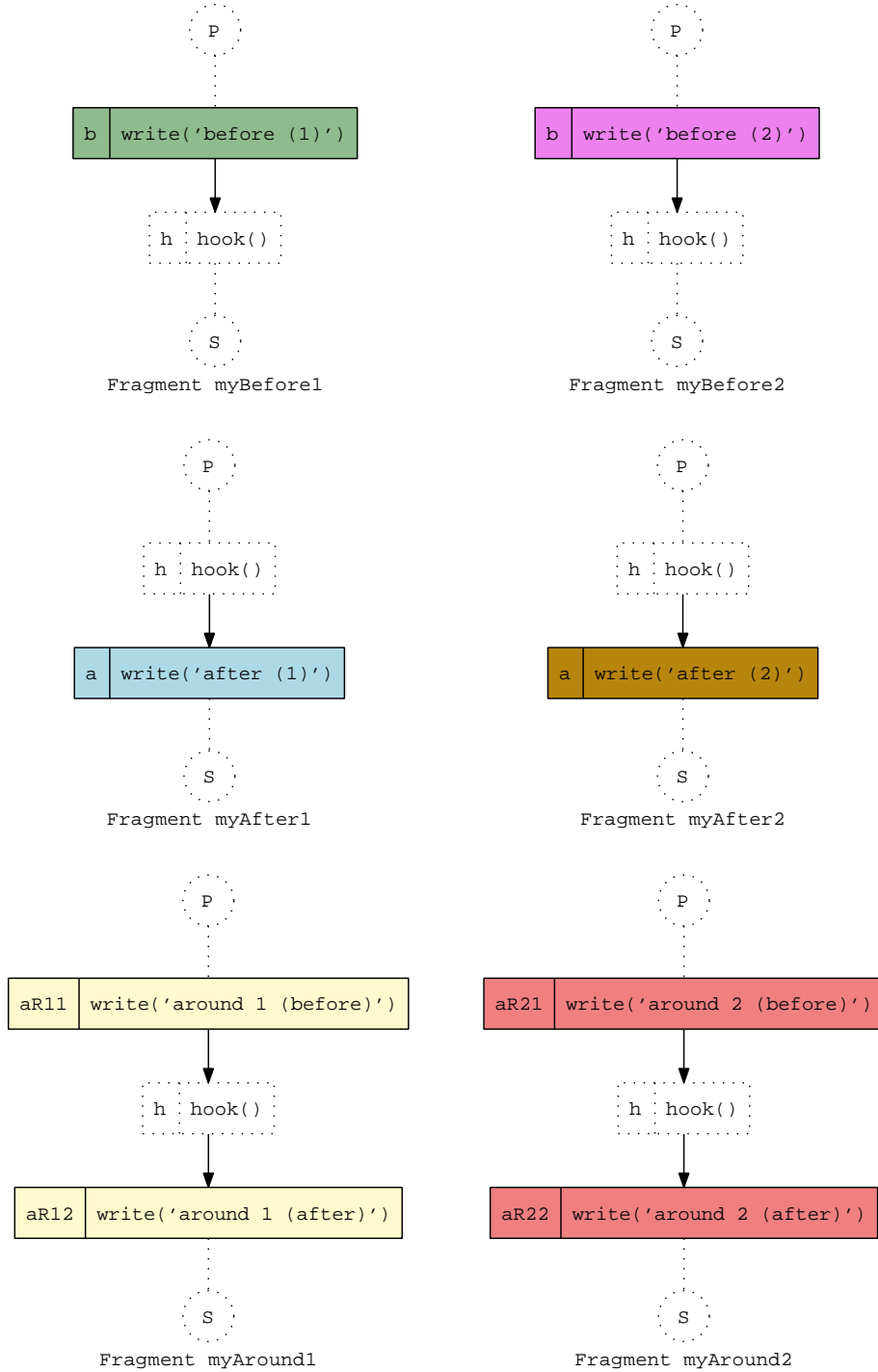


Figure C.2: Fragments used to simulate aspect-ordering in ADORE

All these fragments are intended to be woven on the *do* activity defined in the previous orchestration.

C.2 Default Behavior Defined by ADORE

The usual behavior of ADORE is to identify shared join points, generate a **MERGE** invocation and inform designers of such a choice. As a consequence, the *do* activity is identified as a shared join point, and all the previously defined fragments are merged. We describe in LST C.1 the associated algorithms invocation, at the implementation level. The involved fragments are first cloned (lines 1 and 2). Then, the **MERGE** algorithm is invoked to merge all these fragments (line 3), and finally woven on the targeted activity (line 4). The computed orchestration is depicted in FIG. C.3.

```

1 doClone(myAround1,mar1), doClone(myAround2,mar2), doClone(myBefore1,mb1),
2 doClone(myBefore2,mb2), doClone(myAfter1,ma1), doClone(myAfter2,ma2),
3 doMerge([mar1,mar2,mb1,mb2,ma1,ma2],f), doProcessSimplification(f),
4 doWeave([weave(f,[main_test_do])]),

```

C.3 Simulating “Layered” Weaving

The weave semantics defined by ASPECTJ follows a *layered* approach. Advices defined as **before** are executed first, **around** advices in second, and finally **after** advices in third. According to ASPECTJ semantics, in the absence of an explicit order defined by the programmer, the execution order is undefined. We made the choice to keep parallelism instead of enforcing an order. To simulate such a behavior, consider separately the **around** fragments and the others¹. These two layers are merged independently from each other. Based on these two merged fragments, we simply chain the **WEAVE** invocations to first weave the **around** layer and then weave the other layer. As a consequence, the activities added by the **around** fragments are take into considerations by the second **WEAVE** invocation. In terms of implementation, we describe in LST C.2 the associated algorithms invocation. The obtained process is depicted in FIG. C.4.

```

1 doClone(myAround1,mar1), doClone(myAround2,mar2), doMerge([mar1,mar2],mar),
2 doClone(myBefore1,mb1), doClone(myBefore2,mb2), doClone(myAfter1,ma1),
3 doClone(myAfter2,ma2), doMerge([mb1,mb2,ma1,ma2],f),
4 process:getHook(f,H), doWeave([weave(mar,[H])]), doProcessSimplification(f),
5 doWeave([weave(f,[main_test_do])]),

```

C.4 Introducing Aspect Ordering

ASPECTJ allows programmers to define explicit order between aspects, with the “**declare - precedence**” directive. In this example, we consider the following declaration:

```
declare precedence: myBefore1, myBefore2, myAround1, myAround2, myAfter1, myAfter2
```

To simulate such an explicitly ordered weave, we simply chain the invocations in a more constrained way than the previous one. A precedence declared between *a* and *b* is transformed in ADORE as the weaving of *b* which targets the *hook* defined in *a*. We illustrate this usage in LST C.3. The sequential process obtained as a result is described in FIG. C.5.

```

1 doClone(myAround2,mar2),
2 doClone(myAround1,mar1), process:getHook(mar1,Hmar1), doWeave([weave(mar2,[Hmar1])]),
3 doClone(myBefore2,mb2), process:getHook(mb2,Hmb2), doWeave([weave(mar1,[Hmb2])]),
4 doClone(myBefore1,mb1), process:getHook(mb1,Hmb1), doWeave([weave(mb2,[Hmb1])]),
5 doClone(myAfter2,ma2), process:getHook(ma2,Hma2), doWeave([weave(mb1,[Hma2])]),
6 doClone(myAfter1,ma1), process:getHook(ma1,Hma1), doWeave([weave(ma2,[Hma1])]),
7 doWeave([weave(ma1,[main_test_do])]),

```

¹According the convention used for **before** and **after** fragments, the two layers are implicitly defined

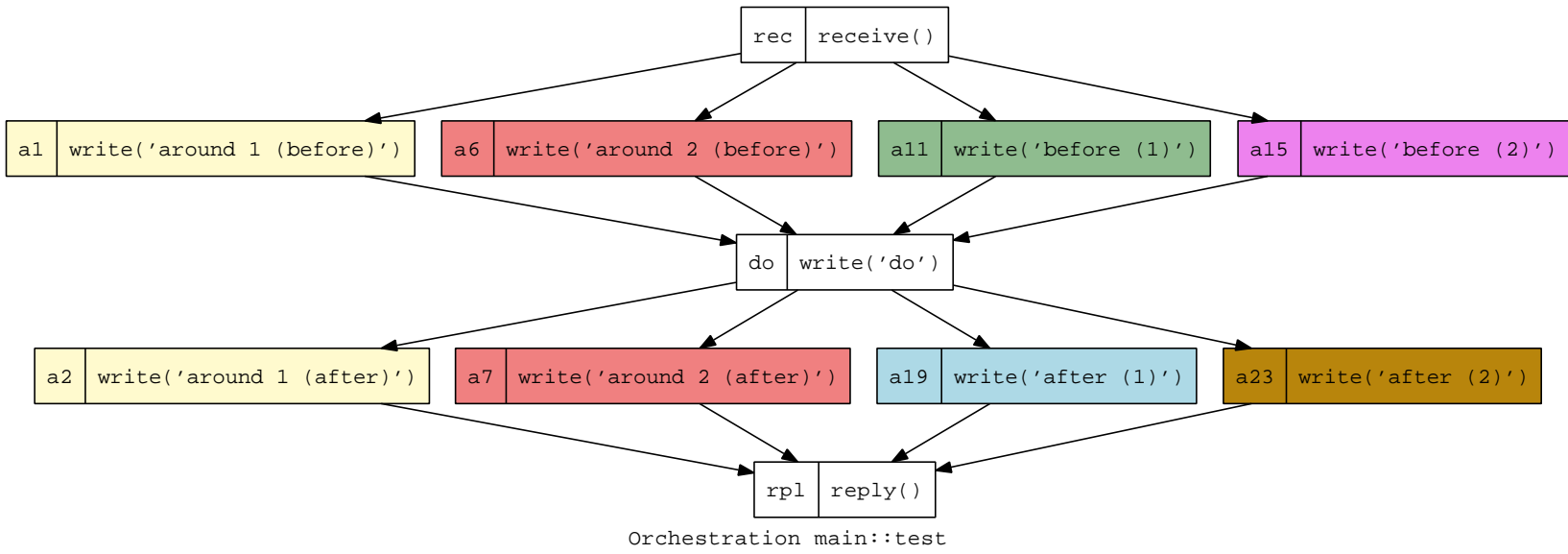
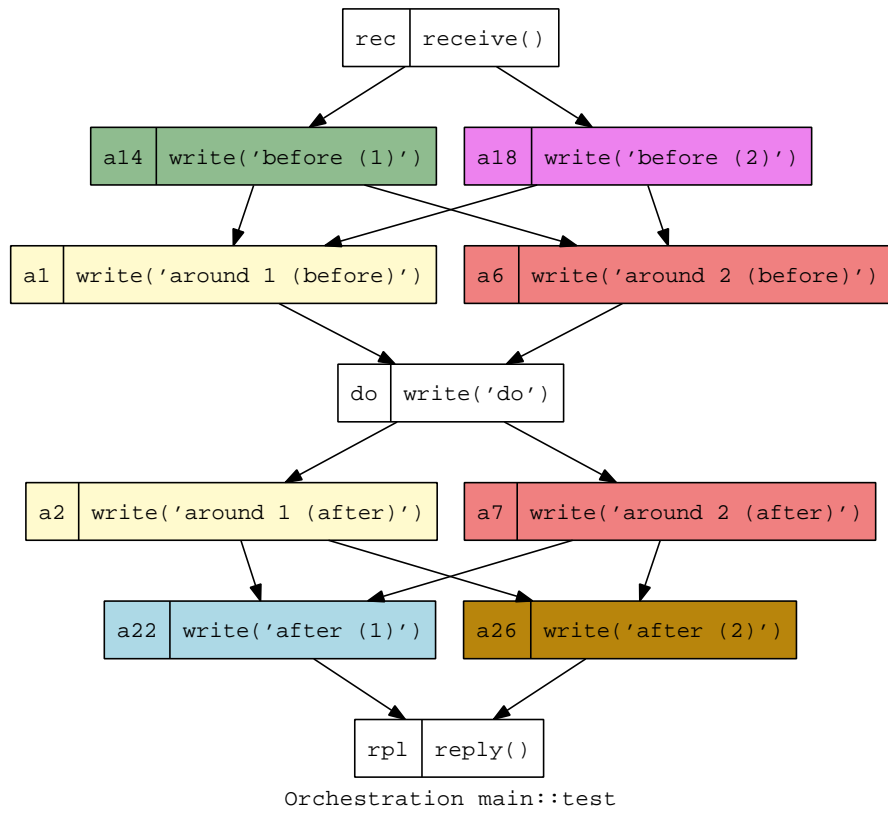


Figure C.3: Result obtained with the usual ADORE MERGE mechanism

Figure C.4: Result obtained with a *layered* mechanism

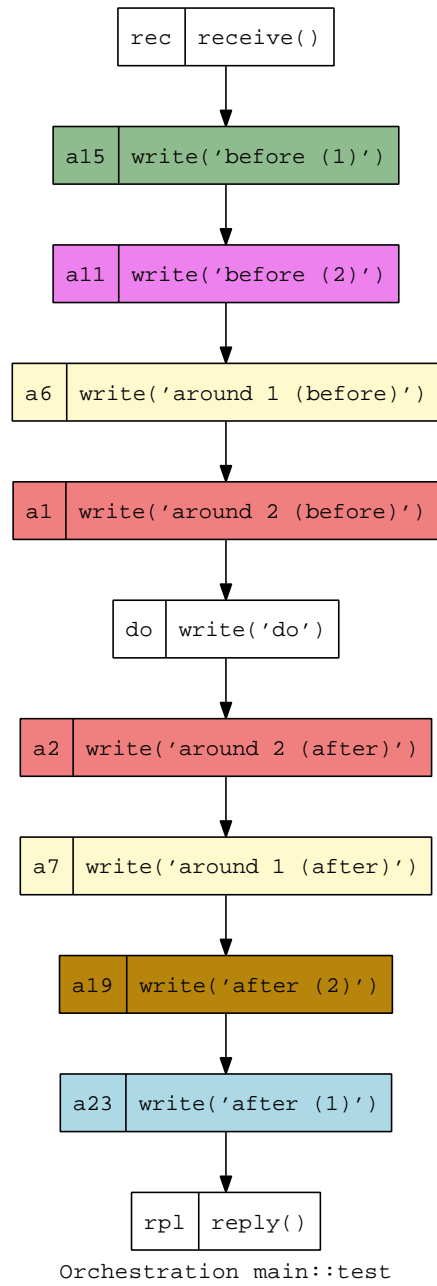


Figure C.5: Process designed with fully-ordered fragment weaving

C.5 ASPECTJ Implementation

In this section, we propose an implementation of this example using the ASPECTJ framework. The initial process is implemented as a Java method, described in LST C.4.

```

1 public class Main {
2     public Main() {}
3     public void myDo() {
4         System.out.println("___do");
5     }
6     public static void main(String[] args) {
7         Main m = new Main();
8         m.myDo();
9     }
10 }

```

Listing C.4: Initial process implemented as a Java method

The different fragments represented in the previous example are implemented as aspects, as described in LST C.5.

```

1 abstract aspect anExtension {
2     declare precedence: myBefore1, myBefore2;
3     declare precedence: myAround2, myAround1;
4     declare precedence: myAfter1, myAfter2;
5
6     pointcut aCall(): execution(* myDo(..)) && this(Main);
7 }
8
9 aspect myBefore1 extends anExtension {
10     before(): aCall() { System.out.println("before_(1)"); }
11 }
12
13 aspect myBefore2 extends anExtension {
14     before(): aCall() { System.out.println("before_(2)"); }
15 }
16
17 aspect myAfter1 extends anExtension {
18     after(): aCall() { System.out.println("after_(1)_"); }
19 }
20
21 aspect myAfter2 extends anExtension {
22     after(): aCall() { System.out.println("after_(2)_"); }
23 }
24
25 aspect myAround1 extends anExtension {
26     void around(): aCall() {
27         System.out.println("___around_1_(before)");
28         proceed();
29         System.out.println("___around_1_(after)");
30     }
31 }
32
33 aspect myAround2 extends anExtension {
34     void around(): aCall() {
35         System.out.println("___around_2_(before)");
36         proceed();
37         System.out.println("___around_2_(after)");
38     }
39 }

```

Listing C.5: Aspects defined to implement the described fragments

Based on these artifacts, we obtain the following execution traces, coherent with the process obtained with ADORE in ordered mode.

```
mosser@necronomicon:ordering/aspectJ$ javac Main.java
mosser@necronomicon:ordering/aspectJ$ java Main
do
mosser@necronomicon:ordering/aspectJ$ ajc Main.java aspects.aj
mosser@necronomicon:ordering/aspectJ$ java Main
before (1)
before (2)
  around 2 (before)
  around 1 (before)
  do
    around 1 (after)
    around 2 (after)
after (2)
after (1)
mosser@necronomicon:ordering/aspectJ$
```



A BPEL implementation of PICWEB

«Few things are harder to put up with than the annoyance of a good example.»

Mark Twain

Contents

D.1 Implementation Overview & Choices	213
D.2 Implementing Processes with the NETBEANS 6.7 IDE	213
D.3 XML Code Associated to PICWEB	213

D.1 Implementation Overview & Choices

In this chapter, we describe the different artifacts needed at the implementation level to realize the PICWEB application as an orchestration of services. This version of the system is implemented with the NETBEANS 6.7 IDE, and deployed in a GLASSFISH 2.2 application server. The business process is implemented as a BPEL orchestration (and its associated WSDL & XSD artifacts), and bundled into a JBI component. This component is deployed into the application server, and handled by the associated BPEL engine.

D.2 Implementing Processes with the NETBEANS 6.7 IDE

The BPEL language can be seen as an assembly language to orchestrate messages exchanges between services. Software editors provide high level graphical IDE to allow a more easier interaction with BPEL codes. We illustrate such editors in FIG. D.1. This screenshot represents the NETBEANS 6.7 IDE displaying a graphical representation associated to the initial version of PICWEB. The left and right columns represent WSDL interfaces provided (left) and consumed(right). The middle part of the editor supports the design of the business process in itself. Nested activities are represented as nested boxes.

However, such a graphical representation suffers from a scalability issue. We provide in FIG. D.2 a screenshot of the same editor while designing the final version of PICWEB. Even of this process is a *simple* one, the readability of the graph and the high level of details make it difficult to read.

D.3 XML Code Associated to PICWEB

We focus now on the code representation associated to the final version of PICWEB. We voluntarily hide the WSDL and XSD artifacts associated to this example for concision reasons. The complete implementation of this business process is available on the project website.

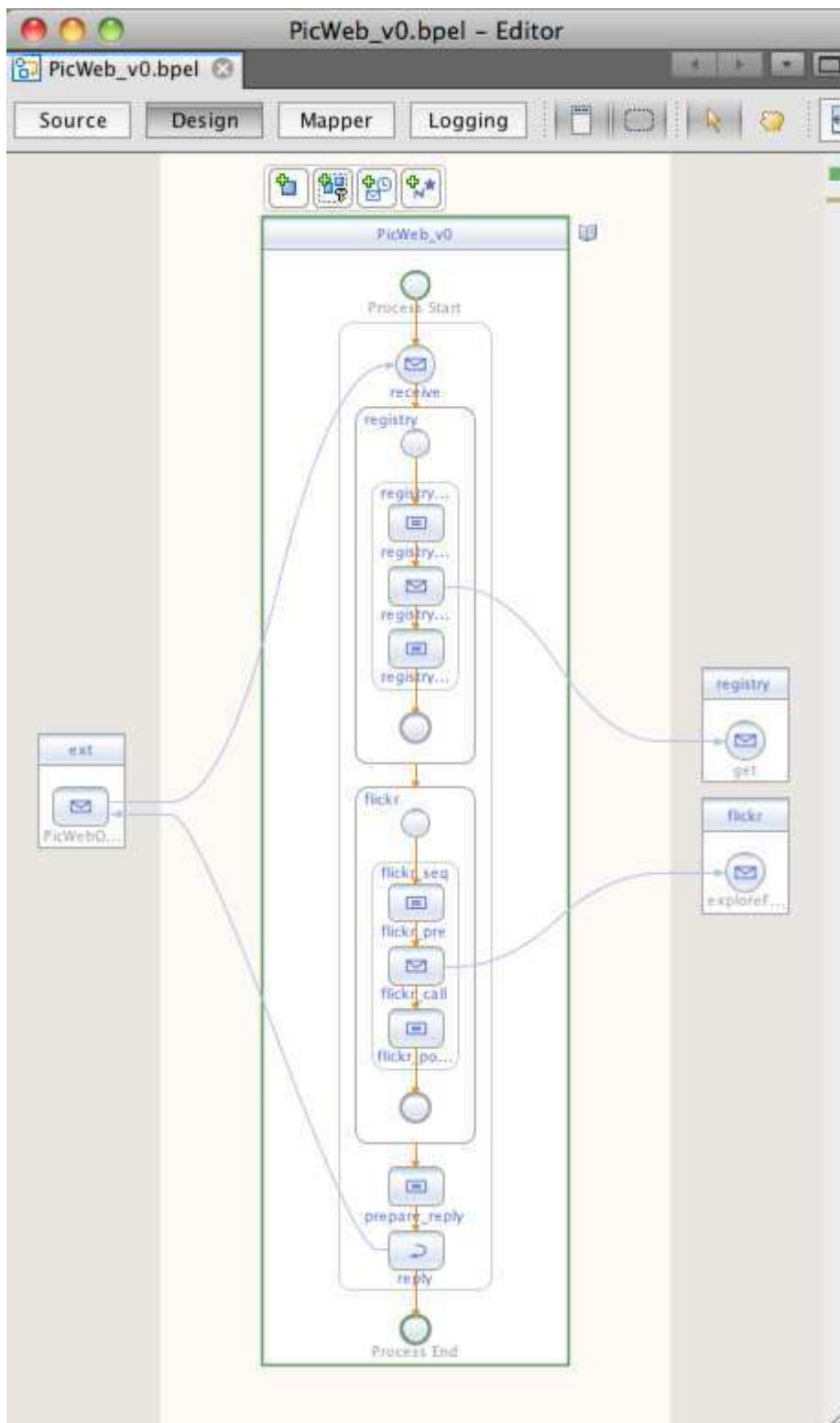


Figure D.1: Initial version of the PICWEB process

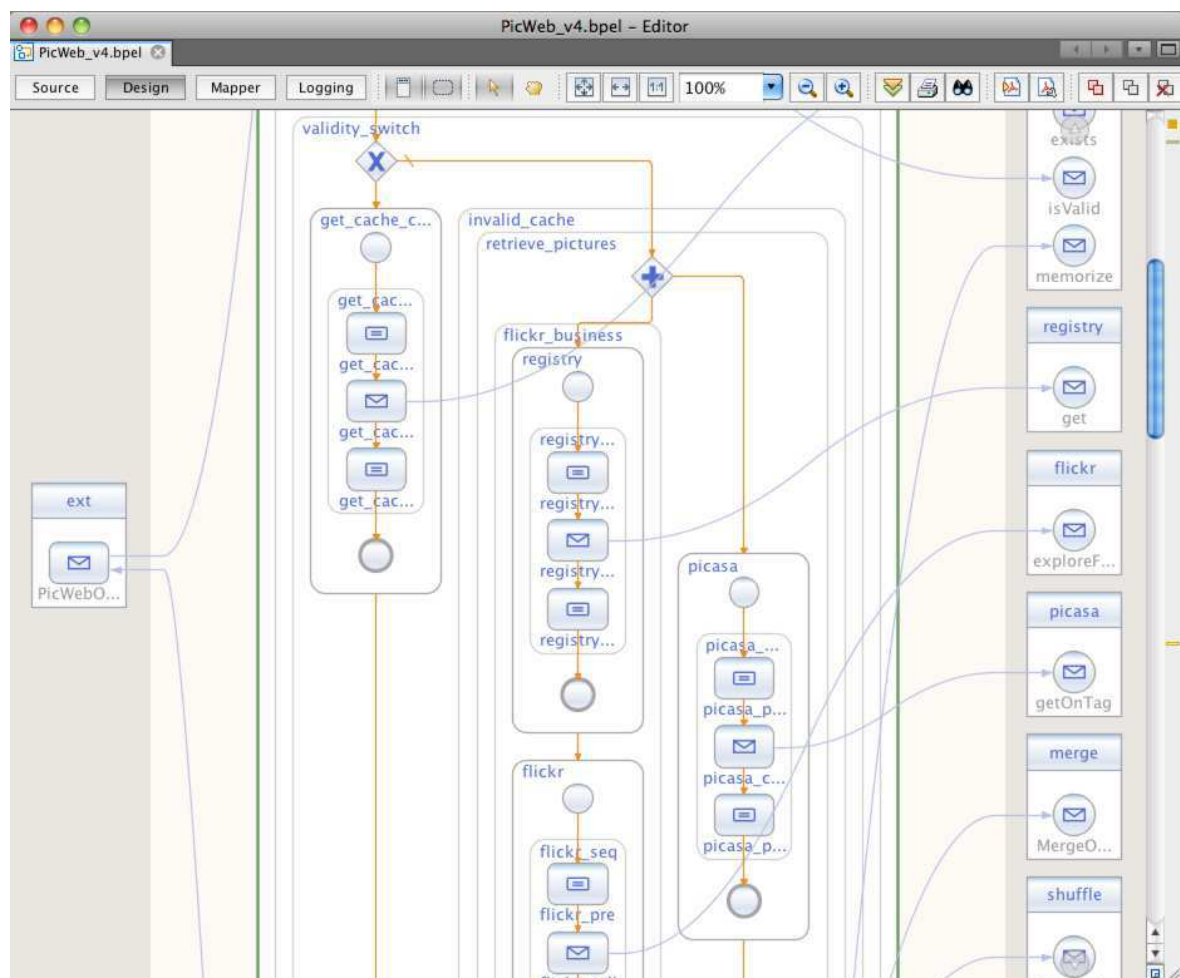


Figure D.2: Final version of the PICWEB process

We provides in LST D.1 an extract of the BPEL code which implements this business process. We hide technical details such as namespace declaration and process wrapping. These 280 lines of code represents the *essence* of the process.

```

1 <partnerLinks>
2   <partnerLink name="cache" partnerLinkType="tns:DataCacheLinkType"
3     partnerRole="DataCacheRole"/>
4   <partnerLink name="registry" partnerLinkType="tns:KeyRegistryLinkType"
5     partnerRole="KeyRegistryRole"/>
6   <partnerLink name="flickr" partnerLinkType="tns:FlickrWrapperLinkType"
7     partnerRole="FlickrWrapperRole"/>
8   <partnerLink name="picasa" partnerLinkType="tns:PicasaBinderLinkType"
9     partnerRole="PicasaBinderRole"/>
10  <partnerLink name="merge" partnerLinkType="tns:Merge"
11    partnerRole="MergePortTypeRole"/>
12  <partnerLink name="shuffle" partnerLinkType="tns:Shuffle"
13    partnerRole="ShufflePortTypeRole"/>
14  <partnerLink name="truncate" partnerLinkType="tns:Truncate"
15    partnerRole="TruncatePortTypeRole"/>
16  <partnerLink name="ext" partnerLinkType="tns:PicWeb" myRole="PicWebPortTypeRole"/>
17 </partnerLinks>
18 <variables>
19   <variable name="shuffled_set" type="ns0:SetOfPictureUrl"/>
20   <variable name="validity" type="xsd:boolean"/>
21   <variable name="merged_set" type="ns0:SetOfPictureUrl"/>
22   <variable name="picasa_set" type="ns0:SetOfPictureUrl"/>
23   <variable name="flickr_set" type="ns0:SetOfPictureUrl"/>
24   <variable name="key" type="xsd:string"/>
25   <variable name="result" type="ns0:SetOfPictureUrl"/>
26   <variable name="PicWebOperationOut" messageType="tns:PicWebOperationResponse"/>
27   <variable name="PicWebOperationIn" messageType="tns:PicWebOperationRequest"/>
28 </variables>
29 <sequence>
30   <receive name="receive" createInstance="yes" partnerLink="ext"
31     operation="PicWebOperation" portType="tns:PicWebPortType"
32     variable="PicWebOperationIn"/>
33   <scope name="cache_validity">
34     <variables>
35       <variable name="IsValidOut" messageType="tns:isValidResponse"/>
36       <variable name="IsValidIn" messageType="tns:isValid"/>
37     </variables>
38     <sequence name="cache_validity_seq">
39       <assign name="cache_validity_pre">
40         <copy>
41           <from>$PicWebOperationIn.in/ns0:tag</from>
42           <to>$IsValidIn.parameters/code</to>
43         </copy>
44         <copy>
45           <from>120</from>
46           <to>$IsValidIn.parameters/delta</to>
47         </copy>
48       </assign>
49       <invoke name="cache_validity_call" partnerLink="cache" operation="isValid"
50         portType="tns:DataCache" inputVariable="IsValidIn"
51         outputVariable="IsValidOut"/>
52       <assign name="cache_validity_post">
53         <copy>
54           <from>$IsValidOut.parameters/return</from>
55           <to variable="validity"/>
56         </copy>
57       </assign>
58     </sequence>
59   </scope>
60   <if name="validity_switch">
61     <condition>$validity</condition>
62     <scope name="get_cache_content">
63       <variables>
64         <variable name="GetValueOut" messageType="tns:getValueResponse"/>
65         <variable name="GetValueIn" messageType="tns:getValue"/>
66       </variables>

```

```

67     <sequence name="get_cache_content_seq">
68         <assign name="get_cache_content_pre">
69             <copy>
70                 <from>$PicWebOperationIn.in/ns0:tag</from>
71                 <to>$GetValueIn.parameters/code</to>
72             </copy>
73         </assign>
74         <invoke name="get_cache_content_call" partnerLink="cache" operation="getValue"
75             portType="tns:DataCache" inputVariable="GetValueIn"
76             outputVariable="GetValueOut"/>
77         <assign name="get_cache_content_post">
78             <copy>
79                 <from>sxxf:doUnMarshal($GetValueOut.parameters/return)</from>
80                 <to variable="merged_set"/>
81             </copy>
82         </assign>
83     </sequence>
84 </scope>
85 <else>
86     <sequence name="invalid_cache">
87         <flow name="retrieve_pictures">
88             <sequence name="flickr_business">
89                 <scope name="registry">
90                     <variables>
91                         <variable name="GetOut" messageType="ns2:getResponse"/>
92                         <variable name="GetIn" messageType="ns2:get"/>
93                     </variables>
94                     <sequence name="registry_seq">
95                         <assign name="registry_pre">
96                             <copy>
97                                 <from>'flickr'</from>
98                                 <to>$GetIn.parameters/name</to>
99                             </copy>
100                        </assign>
101                        <invoke name="registry_call" partnerLink="registry" operation="get"
102                            portType="ns2:KeyRegistry" inputVariable="GetIn"
103                            outputVariable="GetOut"/>
104                        <assign name="registry_post">
105                            <copy>
106                                <from>$GetOut.parameters/return</from>
107                                <to variable="key"/>
108                            </copy>
109                        </assign>
110                    </sequence>
111                </scope>
112                <scope name="flickr">
113                    <variables>
114                        <variable name="ExploreFolksonomyOut"
115                            messageType="ns5:exploreFolksonomyResponse"/>
116                        <variable name="ExploreFolksonomyIn"
117                            messageType="ns5:exploreFolksonomy"/>
118                    </variables>
119                    <sequence name="flickr_seq">
120                        <assign name="flickr_pre">
121                            <copy>
122                                <from>$PicWebOperationIn.in/ns0:tag</from>
123                                <to>$ExploreFolksonomyIn.parameters/search</to>
124                            </copy>
125                            <copy>
126                                <from variable="key"/>
127                                <to>$ExploreFolksonomyIn.parameters/api_key</to>
128                            </copy>
129                        </assign>
130                        <invoke name="flickr_call" partnerLink="flickr" operation="exploreFolksonomy"
131                            portType="ns5:FlickrWrapper" inputVariable="ExploreFolksonomyIn"
132                            outputVariable="ExploreFolksonomyOut"/>
133                        <assign name="flickr_post">
134                            <copy>
135                                <from>$ExploreFolksonomyOut.parameters/return</from>
136                                <to>$flickr_set/ns0:pictureUrl</to>

```



```

137         </copy>
138     </assign>
139 </sequence>
140 </scope>
141 </sequence>
142 <scope name="picasa">
143     <variables>
144         <variable name="GetOnTagOut" messageType="ns4:getOnTagResponse" />
145         <variable name="GetOnTagIn" messageType="ns4:getOnTag" />
146     </variables>
147     <sequence name="picasa_tecj">
148         <assign name="picasa_pre">
149             <copy>
150                 <from>$PicWebOperationIn.in/ns0:tag</from>
151                 <to>$GetOnTagIn.parameters/tag</to>
152             </copy>
153         </assign>
154         <invoke name="picasa_call" partnerLink="picasa" operation="getOnTag"
155             portType="ns4:PicasaBinder" inputVariable="GetOnTagIn"
156             outputVariable="GetOnTagOut" />
157         <assign name="picasa_post">
158             <copy>
159                 <from>$GetOnTagOut.parameters/return</from>
160                 <to>$picasa_set/ns0:pictureUrl</to>
161             </copy>
162         </assign>
163     </sequence>
164 </scope>
165 </flow>
166 <scope name="merge">
167     <variables>
168         <variable name="MergeOperationOut" messageType="ns6:MergeOperationResponse" />
169         <variable name="MergeOperationIn" messageType="ns6:MergeOperationRequest" />
170     </variables>
171     <sequence name="merge_seq">
172         <assign name="merge_pre">
173             <copy>
174                 <from variable="picasa_set" />
175                 <to>$MergeOperationIn.in/ns3:set1</to>
176             </copy>
177             <copy>
178                 <from variable="flickr_set" />
179                 <to>$MergeOperationIn.in/ns3:set2</to>
180             </copy>
181         </assign>
182         <invoke name="merge_call" partnerLink="merge" operation="MergeOperation"
183             portType="ns6:MergePortType" inputVariable="MergeOperationIn"
184             outputVariable="MergeOperationOut" />
185         <assign name="merge_post">
186             <copy>
187                 <from>$MergeOperationOut.out/ns3:result</from>
188                 <to variable="merged_set" />
189             </copy>
190         </assign>
191     </sequence>
192 </scope>
193 <scope name="set_cache_content">
194     <variables>
195         <variable name="MemorizeOut" messageType="tns:memorizeResponse" />
196         <variable name="MemorizeIn" messageType="tns:memorize" />
197     </variables>
198     <sequence name="set_cache_content_seq">
199         <assign name="set_cache_content_pre">
200             <copy>
201                 <from>$PicWebOperationIn.in/ns0:tag</from>
202                 <to>$MemorizeIn.parameters/code</to>
203             </copy>
204             <copy>
205                 <from>sxxf:doMarshal($merged_set)</from>
206                 <to>$MemorizeIn.parameters/data</to>

```

```

207         </copy>
208     </assign>
209     <invoke name="set_cache_content_call" partnerLink="cache" operation="memorize"
210         portType="tns:DataCache" inputVariable="MemorizeIn"
211         outputVariable="MemorizeOut" />
212 </sequence>
213 </scope>
214 </sequence>
215 </else>
216 </if>
217 <scope name="shuffle">
218     <variables>
219         <variable name="ShuffleOperationOut" messageType="tns:ShuffleOperationResponse" />
220         <variable name="ShuffleOperationIn" messageType="tns:ShuffleOperationRequest" />
221     </variables>
222     <sequence name="shuffle_seq">
223         <assign name="shuffle_pre">
224             <copy>
225                 <from variable="merged_set" />
226                 <to>$ShuffleOperationIn.in/ns7:set</to>
227             </copy>
228             <copy>
229                 <from>2</from>
230                 <to>$ShuffleOperationIn.in/ns7:saltScale</to>
231             </copy>
232         </assign>
233         <invoke name="shuffle_call" partnerLink="shuffle" operation="ShuffleOperation"
234             portType="tns:ShufflePortType" inputVariable="ShuffleOperationIn"
235             outputVariable="ShuffleOperationOut" />
236         <assign name="shuffle_post">
237             <copy>
238                 <from>$ShuffleOperationOut.out/ns7:result</from>
239                 <to variable="shuffled_set" />
240             </copy>
241         </assign>
242     </sequence>
243 </scope>
244 <scope name="truncate">
245     <variables>
246         <variable name="TruncateOperationOut" messageType="tns:TruncateOperationResponse" />
247         <variable name="TruncateOperationIn" messageType="tns:TruncateOperationRequest" />
248     </variables>
249     <sequence name="truncate_seq">
250         <assign name="truncate_pre">
251             <copy>
252                 <from>$PicWebOperationIn.in/ns0:threshold</from>
253                 <to>$TruncateOperationIn.in/ns1:threshold</to>
254             </copy>
255             <copy>
256                 <from variable="shuffled_set" />
257                 <to>$TruncateOperationIn.in/ns1:set</to>
258             </copy>
259         </assign>
260         <invoke name="truncate_call" partnerLink="truncate" operation="TruncateOperation"
261             portType="tns:TruncatePortType" inputVariable="TruncateOperationIn"
262             outputVariable="TruncateOperationOut" />
263         <assign name="truncate_post">
264             <copy>
265                 <from>$TruncateOperationOut.out/ns1:set</from>
266                 <to variable="result" />
267             </copy>
268         </assign>
269     </sequence>
270 </scope>
271 <assign name="prepare_reply">
272     <copy>
273         <from variable="result" />
274         <to>$PicWebOperationOut.out/ns0:result</to>
275     </copy>
276 </assign>

```

```
277 <reply name="reply" partnerLink="ext" operation="PicWebOperation"  
278     portType="tns:PicWebPortType" variable="PicWebOperationOut" />  
279 </sequence>
```

Listing D.1: BPEL implementation realizing the PICWEB business process



An ASPECTJ implementation of PICWEB

«I didn't fail the test, I just found 100 ways to do it wrong.»
Benjamin Franklin

Contents

E.1 Initial System	221
E.2 Enhancing PICWEB with ASPECTJ	222
E.2.1 Abstract Aspect to Factorize the Pointcut	222
E.2.2 Enhancements: <code>picasa</code> & <code>truncate</code>	223
E.2.3 Compilation & Execution of the Aspectized System	223
E.2.4 Interference Resolution	224
E.3 Complete Implementation of PICWEB	224

We provide in this chapter an implementation of the PICWEB system, based on the ASPECTJ framework. Part of this work was pair-programmed with Max Kramer during a workshop on Aspect-Oriented Modeling organized by the Mc Gill University. His M.Sc. thesis focuses on the transformation of Reusable Aspect Models into executable programs using ASPECTJ. The complete code associated to this chapter is available on the ADORE code repository (revision 208):

<http://adore.googlecode.com/svn/trunk/experiences/picweb/AspectJ/>

E.1 Initial System

We propose a simulated environment for PICWEB. Legacy PICWEB services are implemented as plain Java classes. We simulate the service usual life-cycle by sharing singleton instances on the different pieces of code. We propose in LST. E.1 an initial implementation of the *getPictures* process. This process consumes two services (**Flickr** & **KeyRegistry**), and basically retrieves the key associated to the FLICKR[®] service before invoking it.

```

1 package picweb;
2 public class PicWeb {
3     public static Flickr FLICKR = new Flickr();
4     public static KeyRegistry REGISTRY = new KeyRegistry();
5     public String[] getPictures(String tag, int threshold) {
6         String key = REGISTRY.getKey("flickr");
7         String[] flickrSet = FLICKR.getPictures(key, tag);
8         return flickrSet;
9     }
10 }

```

Listing E.1: Java implementation of the PICWEB initial process: *getPictures*

Based on this implementation, we define a **Main** class as the entry point of this experience. This class is described in LST E.2, and simply instantiates a **PicWeb** instance before invoking the process to retrieve 3 pictures tagged with the **aTag** information.

```

1 public class Main {
2     public static void main(String[] args) {
3         System.out.println("###_Main:_Calling_PicWeb_###");
4         PicWeb picweb = new PicWeb();
5         String[] data = picweb.getPictures("aTag",3);
6         System.out.println("###_Main:_Displaying_Results_###");
7         for(d: data)
8             System.out.print(d + " ");
9     }
10 }

```

Listing E.2: Entry Point: the **Main** executable class

This code is compiled and executed as the following:

```

mosser@necronomicon:picweb/AspectJ$ javac -version
javac 1.6.0_20
mosser@necronomicon:picweb/AspectJ$ javac picweb/*.java Main.java
mosser@necronomicon:picweb/AspectJ$ java -version
java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02-279-10M3065)
Java HotSpot(TM) 64-Bit Server VM (build 16.3-b01-279, mixed mode)
mosser@necronomicon:picweb/AspectJ$ java Main
### Main: Calling PicWeb ###
### Main: Displaying Results ###
http://flickr/aTag/0/227880700.jpg http://flickr/aTag/1/3318878.jpg
http://flickr/aTag/2/323156151.jpg http://flickr/aTag/3/637702224.jpg
http://flickr/aTag/4/1640207320.jpg http://flickr/aTag/5/958164350.jpg
http://flickr/aTag/6/1161421641.jpg http://flickr/aTag/7/1979648410.jpg
http://flickr/aTag/8/1121449394.jpg
mosser@necronomicon:picweb/AspectJ$

```

E.2 Enhancing PICWEB with ASPECTJ

The initial system needs to be enriched with several concerns to fit the PICWEB requirements. In this chapter, we only describe two extensions: (i) adding PICASA™ as an available picture repository and (ii) truncating the retrieved set to match the given **threshold**.

E.2.1 Abstract Aspect to Factorize the Pointcut

Instead of spreading the pointcut definition into several aspects, we decide to define an abstract aspect (LST E.3) which factorizes the PICWEB pointcut. This pointcut is defined as the interception of the call of a method named **getPictures** targeting a **PicWeb** instance. The two arguments **s** and **l** (respectively defined as **String** and **int**) are also intercepted (lines 6, 7). We use this artifact to factorize the services instances, simulating the usual service life-cycle.

```

1 package foo;
2 import picweb.*;
3
4 abstract aspect factorized {
5
6     protected pointcut callPicWeb(String s, int l):
7         target(PicWeb) && call(String[] getPictures(String, int)) && args(s,l);
8
9     protected Picasa    PICASA    = new Picasa();
10    protected Merge     MERGE     = new Merge();
11    protected Truncate  TRUNCATE  = new Truncate();
12 }

```

Listing E.3: Common aspect used to factorize the PICWEB pointcut and the available services

E.2.2 Enhancements: picasa & truncate

We extend this abstract aspect to implement the two concerns previously cited. The PICASA™ repository is introduced by an aspect named `Picasa`, described in LST E.4. As there is no intrinsic parallelism in Java, we define a sequential implementation: the aspect retrieves the pictures stored in PICASA™, then performs the usual behavior (`proceed`), and sends the two retrieved sets to a service which merges them. The merged set is then returned to the caller.

```

1 package foo;
2 aspect picasa extends factorized
3 {
4     String[] around(String tag, int threshold): callPicWeb(tag,threshold) {
5         System.out.println("==>_around@picasa:_start_<====");
6         String[] picasa = PICASA.explore(tag);
7         String[] raw = proceed(tag,threshold);
8         String[] result = MERGE.run(raw,picasa);
9         System.out.println("==>_around@picasa:_end_<====");
10        return result;
11    }
12 }

```

Listing E.4: Adding the PICASA™ repository with an aspect

The `truncate` aspect implements the following behavior: the usual behavior is called to retrieve the expected picture set, and then sent to a dedicated service which restricts its cardinality.

```

1 package foo;
2 aspect truncate extends factorized
3 {
4     String[] around(String s, int i): callPicWeb(s,i) {
5         System.out.println("==>_around@truncate:_start_<====");
6         String[] raw = proceed(s,i);
7         String[] result = TRUNCATE.run(raw,i);
8         System.out.println("==>_around@truncate:_end_<====");
9         return result;
10    }
11 }

```

Listing E.5: Truncating the retrieved picture set with an aspect

E.2.3 Compilation & Execution of the Aspectized System

We use ASPECTJ to compile the legacy Java classes and the newly added aspects. According to its weave semantic, aspects encountered around shared join points are ordered in an arbitrary way. Under these conditions, the ASPECTJ weaver gives priority to the `picasa` aspect. As a consequence, the behavior of the process is not the expected one: it retrieved only 3 pictures from FLICKR®, but all the pictures stored in PICASA™ are returned.

```

mosser@necronomicon:picweb/AspectJ$ ajc -v
AspectJ Compiler 1.6.9 (1.6.9 - Built: Monday Jul 5, 2010 at 15:28:35 GMT)
- Eclipse Compiler 0.785_R33x, 3.3
mosser@necronomicon:picweb/AspectJ$ ajc foo/*.aj picweb/*.java Main.java
mosser@necronomicon:picweb/AspectJ$ java Main
### Main: Calling PicWeb ###
==> around@picasa: start <====
==> around@truncate: start <====
==> around@truncate: end <====
==> around@picasa: end <====
### Main: Displaying Results ###
http://flickr/aTag/0/2053982697.jpg http://picasa/aTag/0/1921508604.jpg
http://flickr/aTag/1/762924805.jpg http://picasa/aTag/1/1480936115.jpg
http://flickr/aTag/2/1803095015.jpg http://picasa/aTag/2/691756950.jpg
http://picasa/aTag/3/670113902.jpg http://picasa/aTag/4/1376587185.jpg
http://picasa/aTag/5/2080312885.jpg http://picasa/aTag/6/1875826715.jpg
http://picasa/aTag/7/743514902.jpg
mosser@necronomicon:picweb/AspectJ$

```

E.2.4 Interference Resolution

To solve this interference, it is mandatory to explicitly order these two aspects. We add a **declare precedence** declaration in the **factorized** aspect to fulfill this goal. With this rule, the ASPECTJ weaver gives precedence to the **truncate** aspect: it is executed earlier in the aspect chain. Consequently, when entering the **truncate** advice, the **proceed** returned value will contain the PICASA™ pictures (the **picasa** aspect is executed later in the aspect chain). We propose in LST E.6 an enriched version of the **factorized** aspect, declaring this precedence.

```

1  abstract aspect factorized {
2      declare precedence: truncate, picasa; // <== Aspect Ordering Declaration
3
4      protected pointcut callPicWeb(String s, int l):
5          target(PicWeb) && call(String[] getPictures(String, int)) && args(s,l);
6      protected Picasa PICASA = new Picasa();
7      protected Merge MERGE = new Merge();
8      protected Truncate TRUNCATE = new Truncate();
9  }

```

Listing E.6: Ordering aspects to solve undetected interference

```

mosser@necronomicon:picweb/AspectJ$ ajc foo/*.aj picweb/*.java Main.java
mosser@necronomicon:picweb/AspectJ$ java Main
### Main: Calling PicWeb ###
==> around@truncate: start <====
==> around@picasa: start <====
==> around@picasa: end <====
==> around@truncate: end <====
### Main: Displaying Results ###
http://flickr/aTag/0/1625623832.jpg http://picasa/aTag/0/348409446.jpg
http://flickr/aTag/1/1552297148.jpg
mosser@necronomicon:picweb/AspectJ$

```

E.3 Complete Implementation of PICWEB

We propose a complete implementation of the PICWEB running example, with details available on the previously given web site. This implementation contains the several functional concerns: (i) **truncate** and (ii) **picasa**, (iii) a **cache** mechanism and (iv) a **shuffle** invocation to randomize the retrieved picture set.

The **truncate** aspect must be the first one in the aspect chain, as it will perform the final truncation. Then, the **shuffle** aspect must be executed before the **cache** one, to randomize results even when cached. Finally, the **cache** aspect must be executed before the **picasa** one, to include in the cached data the pictures retrieved from PICASA™. Consequently, the only execution order leading to the expected result (one in $4 \times 3 \times 2 \times 1 = 24$) in terms of PICWEB semantic is the following:

```
declare precedence: truncate, shuffle, cache, picasa;
```

Based on this declared precedence rule, the execution of the aspectized code returns the expected values.

```

mosser@necronomicon:picweb/AspectJ$ ajc extensions/*.aj picweb/*.java Main.java
mosser@necronomicon:picweb/AspectJ$ java Main
### Main: Calling PicWeb ###
-> Entering: call(String[] picweb.PicWeb.get(String, int))
==> around@truncate: start <====
==> around@shuffle: start <====
==> around@cache: start <====
-> Entering: call(boolean picweb.Cache.isValid(String, int))
<- Leaving: call(boolean picweb.Cache.isValid(String, int))
==> around@picasa: start <====
-> Entering: call(String[] picweb.Picasa.getOnTag(String))
<- Leaving: call(String[] picweb.Picasa.getOnTag(String))

```

```

-> Entering: call(String picweb.KeyRegistry.get(String))
<- Leaving: call(String picweb.KeyRegistry.get(String))
-> Entering: call(String[] picweb.Flickr.exploreFolksonomy(String, String))
<- Leaving: call(String[] picweb.Flickr.exploreFolksonomy(String, String))
-> Entering: call(String[] picweb.Merge.run(String[], String[]))
<- Leaving: call(String[] picweb.Merge.run(String[], String[]))
===> around@picasa: end <====
-> Entering: call(boolean picweb.Cache.memorize(String, String[]))
<- Leaving: call(boolean picweb.Cache.memorize(String, String[]))
===> around@cache: end <====
-> Entering: call(String[] picweb.Shuffle.run(String[]))
<- Leaving: call(String[] picweb.Shuffle.run(String[]))
===> around@shuffle: end <====
-> Entering: call(String[] picweb.Truncate.run(String[], int))
<- Leaving: call(String[] picweb.Truncate.run(String[], int))
===> around@truncate: end <====
<- Leaving: call(String[] picweb.PicWeb.get(String, int))
### Main: Displaying Results ###
http://flickr/aTag/4/636482563.jpg http://picasa/aTag/10/566071465.jpg
http://picasa/aTag/0/693733048.jpg

```

When the process is called a second time, the **cache** aspect inhibits the execution of the **picasa** one, and consequently the execution of the legacy behavior.

```

mosser@necronomicon:picweb/AspectJ$ java Main
### Main: Calling PicWeb ###
-> Entering: call(String[] picweb.PicWeb.get(String, int))
===> around@truncate: start <====
===> around@shuffle: start <====
===> around@cache: start <====
-> Entering: call(boolean picweb.Cache.isValid(String, int))
<- Leaving: call(boolean picweb.Cache.isValid(String, int))
-> Entering: call(String[] picweb.Cache.getValue(String))
<- Leaving: call(String[] picweb.Cache.getValue(String))
===> around@cache: end <====
-> Entering: call(String[] picweb.Shuffle.run(String[]))
<- Leaving: call(String[] picweb.Shuffle.run(String[]))
===> around@shuffle: end <====
-> Entering: call(String[] picweb.Truncate.run(String[], int))
<- Leaving: call(String[] picweb.Truncate.run(String[], int))
===> around@truncate: end <====
<- Leaving: call(String[] picweb.PicWeb.get(String, int))
### Main: Displaying Results ###
http://picasa/aTag/11/1922537237.jpg http://flickr/aTag/16/1115753675.jpg
http://flickr/aTag/9/1122668184.jpg
mosser@necronomicon:picweb/AspectJ$

```


Bibliography

- [Aksit et al., 2009] Aksit, M., Rensink, A., and Staijen, T. (2009). A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 39–50, New York, NY, USA. ACM.
- [Apel and Batory, 2006] Apel, S. and Batory, D. (2006). When to Use Features and Aspects?: a Case Study. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 59–68, New York, NY, USA. ACM.
- [Apel et al., 2008] Apel, S., Kaestner, C., and Lengauer, C. (2008). Research Challenges in the Tension Between Features and Services. In *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 53–58, New York, NY, USA. ACM.
- [Apel et al., 2010] Apel, S., Kästner, C., Größlinger, A., and Lengauer, C. (2010). Type Safety for Feature-Oriented Product Lines. *Automated Software Engg.*, 17(3):251–300.
- [Baligand and Monfort, 2004] Baligand, F. and Monfort, V. (2004). A Concrete Solution for Web Services Adaptability Using Policies and Aspects. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 134–142, New York, NY, USA. ACM.
- [Balzer, 1991] Balzer, R. (1991). Tolerating Inconsistency. In *ICSE '91: Proceedings of the 13th International Conference on Software Engineering*, pages 158–165, Los Alamitos, CA, USA. IEEE Computer Society Press.
- [Bang-Jensen and Gutin, 2000] Bang-Jensen, J. and Gutin, G. (2000). *Digraphs: Theory, Algorithms and Applications*. Springer Verlag, 1 edition.
- [Baniassad and Clarke, 2004] Baniassad, E. and Clarke, S. (2004). Theme: An Approach for Aspect-Oriented Analysis and Design. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA. IEEE Computer Society.
- [Barais and Duchien, 2005] Barais, O. and Duchien, L. (2005). *SafArchie Studio: An ArgoUML extension to build Safe Architectures*, pages 85–100. Springer. ISBN: 0-387-24589-8.
- [Barais et al., 2008a] Barais, O., Klein, J., Baudry, B., Jackson, A., and Clarke, S. (2008a). Composing Multi-View Aspect Models. In *ICCBSS*, pages 43–52. IEEE Computer Society.
- [Barais et al., 2008b] Barais, O., Lawall, J., Meur, A.-F. L., and Duchien, L. (2008b). *Software Architecture Evolution*, pages 233–262. Springer Verlag.

- [Batory, 2006] Batory, D. (2006). A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 3–35. Springer.
- [Batory, 2007] Batory, D. (2007). From Implementation to Theory in Product Synthesis. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 135–136, New York, NY, USA. ACM.
- [Batory, 2008] Batory, D. (2008). Using Modern Mathematics as an FOSD Modeling Language. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 35–44, New York, NY, USA. ACM.
- [Batory and Geraci, 1997] Batory, D. and Geraci, B. J. (1997). Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, 23:67–82.
- [Batory and O'Malley, 1992] Batory, D. and O'Malley, S. (1992). The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398.
- [Batory et al., 2004] Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30:2004.
- [Baudry et al., 2005] Baudry, B., Fleurey, F., France, R., and Reddy, R. (2005). Exploring the Relationship Between Model composition and Model Transformation. In *In: 7th International Workshop on Aspect-Oriented Modeling, Montego*, page 2.
- [Becht et al., 1999] Becht, M., Gurzki, T., Klarmann, J., and Muscholl, M. (1999). ROPE: Role Oriented Programming Environment for Multiagent Systems. In *CoopIS*, pages 325–333. IEEE Computer Society.
- [Belleau et al., 2008] Belleau, F., Nolin, M.-A. A., Tourigny, N., Rigault, P., and Morissette, J. (2008). Bio2RDF: Towards a Mashup to Build Bioinformatics Knowledge Systems. *Journal of biomedical informatics*.
- [Bézivin and Gerbé, 2001] Bézivin, J. and Gerbé, O. (2001). Towards a Precise Definition of the OMG/MDA Framework. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 273, Washington, DC, USA. IEEE Computer Society.
- [Blanc et al., 2009] Blanc, X., Mougénou, A., Mounier, I., and Mens, T. (2009). Incremental Detection of Model Inconsistencies Based on Model Operations. In van Eck, P., Gordijn, J., and Wieringa, R., editors, *CAiSE*, volume 5565 of *Lecture Notes in Computer Science*, pages 32–46. Springer.
- [Blanc et al., 2008] Blanc, X., Mounier, I., Mougénou, A., and Mens, T. (2008). Detecting Model Inconsistency through Operation-Based Model Construction. In Schäfer, W., Dwyer, M. B., and Gruhn, V., editors, *ICSE*, pages 511–520. ACM.
- [Blanc et al., 2005] Blanc, X., Ramalho, F., and Robin, J. (2005). Metamodel Reuse with MOF. In Briand, L. C. and Williams, C., editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 661–675. Springer.
- [Blay-Fornarino et al., 2009] Blay-Fornarino, M., Ferry, N., Mosser, S., Lavirotte, S., and Tigli, J.-Y. (2009). Démonstrateur de l'application SEDUITE. Technical Report F.4.4, RNTL FAROS.
- [Bolie et al., 2006] Bolie, J., Cardella, M., Blanvalet, S., Juric, M., Carey, S., Chandran, P., Coene, Y., Geminiuc, K., Zirn, M., and Gaur, H. (2006). *BPEL Cookbook: Best Practices for SOA-based Integration and Composite Applications Development*. Packt Publishing.
- [Böllert, 1999] Böllert, K. (1999). On Weaving Aspects. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 301–302, London, UK. Springer-Verlag.

- [Booch, 1986] Booch, G. (1986). Object-Oriented Development. *IEEE Trans. Software Eng.*, 12(2):211–221.
- [Booch et al., 2005] Booch, G., Rumbaugh, J., and Jacobson, I. (2005). *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional.
- [Börger et al., 2010] Börger, E., Cavarra, A., and Riccobene, E. (2010). An ASM Semantics for UML Activity Diagrams. In Rus, T., editor, *Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, pages 293–308. Springer Berlin / Heidelberg. 10.1007/3-540-45499-3_22.
- [Bracha and Cook, 1990] Bracha, G. and Cook, W. (1990). Mixin-Based Inheritance. In *OOP-SLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA. ACM.
- [Brown and Alan, 2004] Brown and Alan, W. (2004). Model Driven Architecture: Principles and Practice. *Software and Systems Modeling (SoSyM)*, 3(4):314–327.
- [Brown et al., 1998] Brown, W. J., Malveau, R. C., and Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley.
- [Bruneton et al., 2004] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2004). An Open Component Model and Its Support in Java. In Crnkovic, I., Stafford, J. A., Schmidt, H. W., and Wallnau, K. C., editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer.
- [Cardoso, 2005] Cardoso, J. (2005). Evaluating the Process Control-Flow Complexity Measure. In *ICWS*, pages 803–804. IEEE Computer Society.
- [Cardoso et al., 2006] Cardoso, J., Mendling, J., Neumann, G., and Reijers, H. A. (2006). A Discourse on Complexity of Process Models. In Eder, J. and Dustdar, S., editors, *Business Process Management Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 117–128. Springer.
- [Carton et al., 2009] Carton, A., Driver, C., Jackson, A., and Clarke, S. (2009). Model-Driven Theme/UML. In [Katz et al., 2009], pages 238–266.
- [Chakrabarti and Liu, 2006] Chakrabarti, D. R. and Liu, S.-M. (2006). Inline Analysis: Beyond Selection Heuristics. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 221–232, Washington, DC, USA. IEEE Computer Society.
- [Charfi and Mezini, 2004] Charfi, A. and Mezini, M. (2004). Aspect-Oriented Web Service Composition with AO4BPEL. In *European Conference on Web Services*, pages 168–182.
- [Charfi and Mezini, 2007] Charfi, A. and Mezini, M. (2007). AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web*, 10:309–344. 10.1007/s11280-006-0016-3.
- [Chitchyan et al., 2005] Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto Alarcon, M., Bakker, J., Tekinerdogan, B., Clarke, S., and Jackson, A. (2005). Survey of Analysis and Design Approaches. Technical report, AOSD-Europe Report ULANC-9.
- [Chung and do Prado Leite, 2009] Chung, L. and do Prado Leite, J. (2009). On non-functional requirements in software engineering. In Borgida, A., Chaudhri, V., Giorgini, P., and Yu, E., editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer Berlin / Heidelberg. 10.1007/978-3-642-02463-4_19.
- [Clemens and Northtop, 2001] Clemens, P. and Northtop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

- [Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. The MIT Press, 2nd revised edition edition.
- [Courbis and Finkelstein, 2005] Courbis, C. and Finkelstein, A. (2005). Towards Aspect Weaving Applications. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 69–77, New York, NY, USA. ACM.
- [Crystal, 1999] Crystal, D. (1999). *The Penguin Dictionary of Language*. Penguin.
- [Curbera et al., 2003] Curbera, F., Khalaf, R., Mukhi, N., Tai, S., and Weerawarana, S. (2003). The Next Step in Web Services. *Commun. ACM*, 46(10):29–34.
- [Czarnecki and Helsen, 2003] Czarnecki, K. and Helsen, S. (2003). Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*.
- [Czarnecki et al., 2002] Czarnecki, K., Osterbye, K., and Völter, M. (2002). Generative Programming. In HernÁandez, J. and Moreira, A., editors, *Object-Oriented Technology ECOOP 2002 Workshop Reader*, volume 2548 of *Lecture Notes in Computer Science*, pages 15–29. Springer Berlin / Heidelberg. 10.1007/3-540-36208-8_2.
- [Dijkstra, 1974] Dijkstra, E. W. (1974). On the Role of Scientific Thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66.
- [Dijkstra, 1997] Dijkstra, E. W. (1997). *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- [Din et al., 2008] Din, G., Eckert, K.-P., and Schieferdecker, I. (2008). A Workload Model for Benchmarking BPEL Engines. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:357–360.
- [Dodani, 2004] Dodani, M. H. (2004). From Objects to Services: A Journey in Search of Component Reuse Nirvana. *Journal of Object Technology*, 3(8):49–54.
- [Douence et al., 2004] Douence, R., Fradet, P., and Südholt, M. (2004). Composition, Reuse and Interaction Analysis of Stateful Aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA. ACM.
- [Douence et al., 2006] Douence, R., Le Botlan, D., Noyé, J., and Südholt, M. (2006). Concurrent Aspects. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 79–88, New York, NY, USA. ACM.
- [Dumas and ter Hofstede, 2001] Dumas, M. and ter Hofstede, A. (2001). UML Activity Diagrams as a Workflow Specification Language. In Gogolla, M. and Kobryn, C., editors, *«UML» 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin / Heidelberg. 10.1007/3-540-45441-1_7.
- [Dutra et al., 2004] Dutra, I., Page, D., Costa, V. S., Shavlik, J., and Waddell, M. (2004). Toward Automatic Management of Embarrassingly Parallel Applications. In Kosch, H., Böszörményi, L., and Hellwagner, H., editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 509–516. Springer Berlin / Heidelberg. 10.1007/978-3-540-45209-6_73.
- [Elrad et al., 2001] Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., and Ossher, H. (2001). Discussing Aspects of AOP. *Commun. ACM*, 44(10):33–38.
- [Emmerich et al., 2005] Emmerich, W., Butchart, B., Chen, L., Wassermann, B., and Price, S. (2005). Grid Service Orchestration Using the Business Process Execution Language (BPEL). *Journal of Grid Computing*, 3:283–304. 10.1007/s10723-005-9015-3.

- [ERCIM, 2010] ERCIM (2010). ERCIM Working Group on Software Evolution Terminology. Technical report, ERCIM.
- [Ericsonn, 2000] Ericsonn, M. (2000). Activity Diagrams, What it is and How to Use. http://sunset.usc.edu/classes/cs577a_2000/papers/ActivitydiagramsforRoseArchitect.pdf.
- [Eshuis, 2002] Eshuis, H. (2002). *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, Univ. of Twente.
- [Eshuis and Wieringa, 2001] Eshuis, R. and Wieringa, R. (2001). A Comparison of Petri Net and Activity Diagram Variants. In *Proc. of 2nd Int. Coll. on Petri Net Technologies for Modelling Communication Based Systems*, pages 93–104.
- [Falleri et al., 2006] Falleri, J.-R., Huchard, M., and Nebut, C. (2006). Towards a Traceability Framework for Model Transformations in Kermeta. In *In: ECMDA-TW Workshop*.
- [Fielding, 2000] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California.
- [Fleurey, 2006] Fleurey, F. (2006). *Langage et méthode pour une ingénierie des modèles fiable*. PhD thesis, Université de Rennes 1.
- [Fleurey et al., 2007] Fleurey, F., Baudry, B., France, R., and Ghosh, S. (2007). A Generic Approach For Automatic Model Composition. In *Aspect Oriented Modelling workshop at MoD-ELS*.
- [Fowler, 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [Fowler, 2010] Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional.
- [France and Rumpe, 2007] France, R. and Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA. IEEE Computer Society.
- [France et al., 2006] France, R. B., Ghosh, S., Dinh-Trong, T., and Solberg, A. (2006). Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer*, 39:59–66.
- [Freeman et al., 2008] Freeman, G., Batory, D., and Lavender, G. (2008). Lifting Transformational Models of Product Lines: A Case Study. In Vallecillo, A., Gray, J., and Pierantonio, A., editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin / Heidelberg. 10.1007/978-3-540-69927-9_2.
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.
- [Garlan and Schmerl, 2004] Garlan, D. and Schmerl, B. (2004). Using Architectural Models at Runtime: Research Challenges. In *European Workshop on Software Architectures (EWSA)*, pages 200–205. Springer-Verlag.
- [Gasevic et al., 2006] Gasevic, D., Djuric, D., Devedzic, V., and Selic, B. (2006). *Model Driven Architecture and Ontology Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- [Glatard, 2007] Glatard, T. (2007). *Description, Deployment and Optimization of Medical Image Analysis Workflows on Production Grids*. PhD thesis, Université de Nice Sophia-Antipolis, Sophia Antipolis, France.
- [Gregory, 1997] Gregory, S. (1997). A Declarative Approach to Concurrent Programming. In *PLILP '97: Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 79–93, London, UK. Springer-Verlag.

- [Griss, 2000] Griss, M. L. (2000). Implementing Product-Line Features by Composing Aspects. In *Proceedings of the first conference on Software product lines : experience and research directions*, pages 271–288, Norwell, MA, USA. Kluwer Academic Publishers.
- [Harrison and Ossher, 1993] Harrison, W. and Ossher, H. (1993). Subject-Oriented Programming: a Critique of Pure Objects. *SIGPLAN Not.*, 28(10):411–428.
- [Heckel et al., 2002] Heckel, R., Küster, J. M., and Taentzer, G. (2002). Confluence of Typed Attributed Graph Transformation Systems. In *ICGT'02: Proceedings of the First International Conference on Graph Transformation*, pages 161–176, London, UK. Springer-Verlag.
- [Heidenreich et al., 2009] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., and Wende, C. (2009). Derivation and refinement of textual syntax for models. In Paige, R., Hartman, A., and Rensink, A., editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin / Heidelberg. 10.1007/978-3-642-02674-4_9.
- [Hellerman, 1964] Hellerman, H. (1964). Experimental Personalized Array Translator System. *Commun. ACM*, 7(7):433–438.
- [Henning, 2006] Henning, M. (2006). The Rise and Fall of CORBA. *Queue*, 4(5):28–34.
- [Hinz et al., 2005] Hinz, S., Schmidt, K., and Stahl, C. (2005). Transforming BPEL to Petri Nets. In van der Aalst, W. M. P., Benatallah, B., Casati, F., and Curbera, F., editors, *Proceedings of the 3rd Int'l Conference on Business Process Management (BPM 2005)*, pages 220–235, Nancy, France. Springer Verlag.
- [Hoare, 1978] Hoare, C. A. R. (1978). Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677.
- [Hopcroft et al., 2001] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [Hotho et al., 2006] Hotho, A., Jaschke, R., Schmitz, C., and Stumme, G. (2006). Information Retrieval in Folksonomies: Search and Ranking. *The Semantic Web: Research and Applications*, pages 411–426.
- [Hürsch et al., 1995] Hürsch, W., Lopes, C., Hürsch, W. L., and Lopes, C. V. (1995). Separation of Concerns. Technical report, Computer Science Dept., Northeastern Univ., Boston.
- [ISO, 2003] ISO (2003). *ISO/IEC 14882:2003: Programming languages: C++*. ISO.
- [Iverson, 1962] Iverson, K. E. (1962). *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA.
- [Jackson, 2006] Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 1 edition.
- [Jayaraman et al., 2007] Jayaraman, P. K., Whittle, J., Elkhodary, A. M., and Gomaa, H. (2007). Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In Engels, G., Opdyke, B., Schmidt, D. C., and Weil, F., editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 151–165. Springer.
- [Jouault and Kurtev, 2006] Jouault, F. and Kurtev, I. (2006). Transforming Models with ATL. In Bruel, J.-M., editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin / Heidelberg. 10.1007/11663430_14.
- [Karastoyanova and Leymann, 2009] Karastoyanova, D. and Leymann, F. (2009). BPEL'n'Aspects: Adapting Service Orchestration Logic. In *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*, pages 222–229, Washington, DC, USA. IEEE Computer Society.

- [Kästner et al., 2009] Kästner, C., Apel, S., Trujillo, S., Kuhlemann, M., and Batory, D. (2009). Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *In Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33 of *LNBI*, pages 174–194. Springer-Verlag.
- [Katz and Katz, 2008] Katz, E. and Katz, S. (2008). Incremental Analysis of Interference Among Aspects. In *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, pages 29–38, New York, NY, USA. ACM.
- [Katz et al., 2009] Katz, S., Ossher, H., France, R., and Jézéquel, J.-M., editors (2009). *Transactions on Aspect-Oriented Software Development VI, Special Issue on Aspects and Model-Driven Engineering*, volume 5560 of *Lecture Notes in Computer Science*. Springer.
- [Kellens et al., 2006] Kellens, A., Mens, K., Brichau, J., and Gybels, K. (2006). Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 501–525. Springer-Verlag.
- [Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK. Springer-Verlag.
- [Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In Akşit, M. and Matsuoka, S., editors, *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, chapter 10, pages 220–242. Springer-Verlag, Berlin/Heidelberg.
- [Kienzle et al., 2009a] Kienzle, J., Al Abed, W., and Klein, J. (2009a). Aspect-Oriented Multi-View Modeling. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA. ACM.
- [Kienzle et al., 2009b] Kienzle, J., Duala-Ekoko, E., and Gélneau, S. (2009b). AspectOptima: A Case Study on Aspect Dependencies and Interactions. *T. Aspect-Oriented Software Development*, 5:187–234.
- [Kienzle et al., 2009c] Kienzle, J., Guelfi, N., and Mustafiz, S. (2009c). Crisis Management Systems, A Case Study for Aspect-Oriented Modeling. Requirements Document for TAOSD Special Issue, McGill University & University of Luxembourg.
- [Kim et al., 2008] Kim, C. H. P., Kästner, C., and Batory, D. (2008). On the Modularity of Feature Interactions. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 23–34, New York, NY, USA. ACM.
- [Klein, 2006] Klein, J. (2006). *Aspects Comportementaux et Tissage*. PhD thesis, Université Rennes 1.
- [Klein et al., 2007] Klein, J., Fleurey, F., and Jézéquel, J.-M. (2007). Weaving Multiple Aspects in Sequence Diagrams. *T. Aspect-Oriented Software Development*, 3:167–199.
- [Klein et al., 2006] Klein, J., Hélouet, L., and Jézéquel, J.-M. (2006). Semantic-based Weaving of Scenarios. In *proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany. ACM.
- [Klein and Kienzle, 2007] Klein, J. and Kienzle, J. (2007). Reusable Aspect Models. In *11th Workshop on Aspect-Oriented Modeling, AOM at Models'07*,.
- [Kloppmann et al., 2009] Kloppmann, M., König, D., and Moser, S. (2009). *The Dichotomy of Modeling and Execution: BPMN and WS-BPEL*, chapter 4, pages 70–91. Information Science Reference.

- [Kohlbecker et al., 1986] Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B. (1986). Hygienic Macro Expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA. ACM.
- [Kolovos et al., 2009] Kolovos, D. S., Rose, L. M., Paige, R. F., and Polack, F. A. (2009). Raising the Level of Abstraction in the Development of GMF-Based Graphical Model Editors. *Modeling in Software Engineering, International Workshop on*, 0:13–19.
- [Kozaczynski and Booch, 1998] Kozaczynski, W. and Booch, G. (1998). Guest editors' introduction: Component-based software engineering. *IEEE Software*, 15(5):34–36.
- [Kühne, 2006] Kühne, T. (2006). Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5(4):369–385.
- [Küster and Kunig-Ries, 2006] Küster, U. and Kunig-Ries, B. (2006). Dynamic Binding for BPEL Processes – A Lightweight Approach to Integrate Semantics into Web Services. In *Service-Oriented Computing ICSOC 2006*, volume 4652 of *Lecture Notes in Computer Science*, pages 116–127. Springer Berlin / Heidelberg. 10.1007/978-3-540-75492-3_11.
- [Lagaisse and Joosen, 2006] Lagaisse, B. and Joosen, W. (2006). Decomposition Into Elementary Pointcuts: a Design Principle for Improved Aspect Reusability. In *Software Engineering Properties of Languages and Aspect Technologies*.
- [Laue and Gruhn, 2006] Laue, R. and Gruhn, V. (2006). Complexity Metrics for Business Process Models. In Abramowicz, W. and Mayr, H. C., editors, *BIS*, volume 85 of *LNI*, pages 1–12. GI.
- [Linthicum, 2000] Linthicum, D. S. (2000). *Enterprise Application Integration*. Addison-Wesley Longman Ltd., Essex, UK, UK.
- [Liu and Batory, 2004] Liu, J. and Batory, D. (2004). Automatic Remodularization and Optimized Synthesis of Product-Families. In *In: Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE 2004)*, pages 379–395. Springer-Verlag.
- [Liu et al., 2005] Liu, J., Batory, D. S., and Nedunuri, S. (2005). Modeling Interactions in Feature Oriented Software Designs. In Reiff-Marganiec, S. and Ryan, M., editors, *FIW*, pages 178–197. IOS Press.
- [Lopez-Herrejon et al., 2006] Lopez-Herrejon, R., Batory, D., and Lengauer, C. (2006). A Disciplined Approach to Aspect Composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA. ACM.
- [Magee and Krammer, 2006] Magee, J. and Krammer, J. (2006). *Concurrency: State Models & Java Programs*. wiley, 2 edition.
- [Mallet et al., 2010] Mallet, F., Deantoni, J., André, C., and De Simone, R. (2010). The Clock Constraint Specification Language for building timed causality models. *Innovations in Systems and Software Engineering*, 6:99–106.
- [Masuhara et al., 2003] Masuhara, H., Kiczales, G., and Dutchyn, C. (2003). A Compilation and Optimization Model for Aspect-Oriented Programs. In Hedin, G., editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin / Heidelberg. 10.1007/3-540-36579-6_4.
- [Matougui and Beugnard, 2005] Matougui, S. and Beugnard, A. (2005). How to Implement Software Connectors? A Reusable, Abstract and Adaptable Connector. In Kutvonen, L. and Alonistioti, N., editors, *DAIS*, volume 3543 of *Lecture Notes in Computer Science*, pages 83–94. Springer.
- [McAllester, 1994] McAllester, D. (1994). Variational Attribute Grammars for Computer Aided Design (Release 3.0). Technical report, MIT.

- [McNeile and Roubtsova, 2008] McNeile, A. and Roubtsova, E. (2008). CSP Parallel Composition of Aspect Models. In *AOM '08: Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling*, pages 13–18, New York, NY, USA. ACM.
- [McNeile and Roubtsova, 2010] McNeile, A. and Roubtsova, E. (2010). Aspect-Oriented Development Using Protocol Modeling. *Transactions on Aspect-Oriented Software Development (TAOSD) Special issue on Aspect Oriented Modeling*, pages 1–34.
- [McNeile and Simons, 2006] McNeile, A. T. and Simons, N. (2006). Protocol Modelling: a Modelling Approach that Supports Reusable Behavioural Abstractions. *Software and System Modeling*, 5(1):91–107.
- [Medvidovic et al., 1997] Medvidovic, N., Oreizy, P., and Taylor, R. N. (1997). Reuse of Off-the-Shelf Components in Cs-Style Architectures. *Software Engineering, International Conference on*, 0:692.
- [Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. N. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93.
- [Mellor and Balcer, 2002] Mellor, S. J. and Balcer, M. (2002). *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. Foreword By-Jacobson, Ivar.
- [Merks et al., 2003] Merks, E., Eliersick, R., Grose, T., Budinsky, F., and Steinberg, D. (2003). *The Eclipse Modeling Framework*. Addison Wesley.
- [Miller and Mukerji, 2003] Miller, J. and Mukerji, J. (2003). MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG).
- [Montagnat et al., 2006] Montagnat, J., Glatard, T., and Lingrand, D. (2006). Data Composition Patterns in Service-Based Workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, Paris, France.
- [Montagnat et al., 2009] Montagnat, J., Isnard, B., Glatard, T., Maheshwari, K., and Blay-Fornarino, M. (2009). A Data-Driven Workflow Language for Grids Based on Array Programming Principles. In *WORKS '09: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, New York, NY, USA. ACM.
- [Moreau et al., 2009] Moreau, A., Malenfant, J., and Dao, M. (2009). Data Flow Repair in Web Service Orchestration at Runtime. In Perry, M., Sasaki, H., Ehmann, M., Bellot, G. O., and Dini, O., editors, *ICIW*, pages 43–48. IEEE Computer Society.
- [Mosser et al., 2010a] Mosser, S., Bergel, A., and Blay-Fornarino, M. (2010a). Visualizing and Assessing a Compositional Approach of Business Process Design. In *Software Composition*, , Malaga, Spain. ACM SIGPLAN and SIGSOFT, Springer's Lecture Notes in Computer Sciences.
- [Mosser et al., 2010b] Mosser, S., Blay-Fornarino, M., and France, R. (2010b). Workflow Design using Fragment Composition (Crisis Management System Design through ADORE). *Transactions on Aspect-Oriented Software Development (TAOSD) Special issue on Aspect Oriented Modeling*, pages 1–34.
- [Mosser et al., 2009a] Mosser, S., Blay-Fornarino, M., and Montagnat, J. (2009a). Orchestration Evolution Following Dataflow Concepts: Introducing Unanticipated Loops Inside a Legacy Workflow. In *International Conference on Internet and Web Applications and Services(ICIW), long paper*, pages 389–394, Venice, Italy. IEEE Computer Society.
- [Mosser et al., 2008a] Mosser, S., Blay-Fornarino, M., and Riveill, M. (2008a). Web Services Orchestration Evolution: A Merge Process For Behavioral Evolution. In *2nd European Conference on Software Architecture (ECSA'08)*, pages 35–49, Paphos, Cyprus. Springer LNCS.

- [Mosser et al., 2009b] Mosser, S., Blay-Fornarino, M., and Riveill, M. (2009b). Service Oriented Architecture Definition Using Composition of Business-Driven Fragments. In *Models and Evolution (MODSE'09), workshop*, pages 1–10, Denver, USA.
- [Mosser et al., 2008b] Mosser, S., Chauvel, F., Blay-Fornarino, M., and Riveill, M. (2008b). Web Service Composition: Mashups Driven Orchestration Definition. In *International Conference on Intelligent Agents, Web Technologies and Internet Commerce(IAWTIC'08), long paper*, , pages 284 – 289, Vienna, Austria. IEEE Computer Society.
- [Mosser et al., 2011] Mosser, S., Mussbacher, G., Blay-Fornarino, M., and Amyot, D. (2011). From Aspect-oriented Requirements Models to Aspect-oriented Business Process Design Models. In *10th international conference on Aspect Oriented Software Development(AOSD'11), long paper (1st round)*, , Porto de Galinhas. ACM.
- [Mougenot et al., 2009] Mougenot, A., Blanc, X., and Gervais, M.-P. (2009). D-Praxis : A Peer-to-Peer Collaborative Model Editing Framework. In Senivongse, T. and Oliveira, R., editors, *DAIS*, volume 5523 of *Lecture Notes in Computer Science*, pages 16–29.
- [Mougin and Ducasse, 2003] Mougin, P. and Ducasse, S. (2003). OOPAL: Integrating Array Programming in Object-Oriented Programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 65–77, New York, NY, USA. ACM Press.
- [Muller et al., 2005] Muller, P.-A., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., Studer, P., and Jézéquel, J.-M. (2005). On Executable Meta-Languages applied to Model Transformations. In *Model Transformations In Practice Workshop*, Montego Bay Jamaïque.
- [Muller et al., 2009] Muller, P.-A., Fondement, F., and Baudry, B. (2009). Modeling Modeling. In Schürr, A. and Selic, B., editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 2–16. Springer.
- [Muller et al., 2008] Muller, P.-A., Fondement, F., Fleurey, F., Hassenforder, M., Schneckenburger, R., Gérard, S., and Jézéquel, J.-M. (2008). Model-Driven Analysis and Synthesis of Textual Concrete Syntax. *Software and System Modeling*, 7(4):423–441.
- [Muller and Hassenforder, 2005] Muller, P.-A. and Hassenforder, M. (2005). HUTN as a Bridge between ModelWare and GrammarWare. In *WISME Workshop, MODELS / UML'2005*, Montego Bay, Jamaica.
- [Mussbacher and Amyot, 2009] Mussbacher, G. and Amyot, D. (2009). Extending the User Requirements Notation with Aspect-oriented Concepts. In Reed, R., Bilgic, A., and Gotzhein, R., editors, *SDL 2009: Design for Motes and Mobiles*, volume 5719 of *Lect. Notes Comp. Sci.*, pages 115–132. Springer.
- [Mussbacher et al., 2010] Mussbacher, G., Amyot, D., Araújo, J., and Moreira, A. (2010). Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN): A Case Study. In Katz, S., Mezini, M., and Kienzle, J., editors, *Transactions on Aspect-Oriented Software Development VII*, volume 6210 of *Lect. Notes Comp. Sci.*, pages 23–68. Springer.
- [Mussbacher et al., 2009] Mussbacher, G., Whittle, J., and Amyot, D. (2009). Semantic-Based Interaction Detection in Aspect-Oriented Scenarios. In *RE*, pages 203–212. IEEE Computer Society.
- [Nagy et al., 2005] Nagy, I., Bergmans, L., and Aksit, M. (2005). Composing Aspects at Shared Join Points. In Hirschfeld, R., Kowalczyk, R., Polze, A., and Weske, M., editors, *NODE / GSEM*, volume 69 of *LNI*, pages 19–38. GI.
- [Nejati et al., 2007] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., and Zave, P. (2007). Matching and Merging of Statecharts Specifications. In *In 29th International Conference on Software Engineering (ICSE'07)*, pages 54–64.

- [Nemo et al., 2007] Nemo, C., Glatard, T., Blay-Fornarino, M., and Montagnat, J. (2007). Merging Overlapping Orchestrations: an Application to the Bronze Standard Medical Application. In *International Conference on Services Computing (SCC 2007)*, , pages 364–371. IEEE Computer Engineering.
- [OASIS, 2004] OASIS (2004). UDDI Version 3.0.2. Technical report, OASIS. <http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm>.
- [OASIS, 2006a] OASIS (2006a). Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS.
- [OASIS, 2006b] OASIS (2006b). Web Services Security: SOAP Message Security 1.1. Technical report, OASIS. <http://www.oasis-open.org/committees/download.php/16790/wss-v1.1.1-spec-os-SOAPMessageSecurity.pdf>.
- [OASIS, 2007] OASIS (2007). Web Services Business Process Execution Language Version 2.0. Technical report, OASIS.
- [Oinn et al., 2004] Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M. R., Wipat, A., and Li, P. (2004). Taverna: a Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, 20(17):3045–3054.
- [OMG, 2005] OMG (2005). UML 2.0 Superstructure Specification. Technical report, Object Management Group (OMG).
- [OMG, 2006a] OMG (2006a). Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG).
- [OMG, 2006b] OMG (2006b). CORBA Component Model 4.0 Specification. Specification Version 4.0, Object Management Group.
- [OMG, 2006] OMG (2006). *Meta Object Facility (MOF) Core Specification Version 2.0*.
- [OMG, 2007] OMG (2007). *XML Metadata Interchange (XMI)*. OMG.
- [OpenSOA, 2007] OpenSOA (2007). *SCA Service Component Architecture - Assembly Model Specification*. Open SOA. Version 1.00.
- [Oquendo, 2004] Oquendo, F. (2004). π -ADL: an Architecture Description Language Based on the Higher-Order Typed π -calculus for Specifying Dynamic and Mobile Software Architectures. *SIGSOFT Softw. Eng. Notes*, 29(3):1–14.
- [Ortiz and Leymann, 2006] Ortiz, G. and Leymann, F. (2006). Combining WS-Policy and Aspect-Oriented Programming. *Advanced International Conference on Telecommunications / Internet and Web Applications and Services, International Conference on*, 0:143.
- [Ouyang et al., 2006] Ouyang, C., Dumas, M., ter Hofstede, A. H., and van der Aalst, W. M. (2006). From BPMN Process Models to BPEL Web Services. *Web Services, IEEE International Conference on*, 0:285–292.
- [Pant, 2008] Pant, K. (2008). *Business Process Driven SOA using BPMN and BPEL: From Business Process Modeling to Orchestration and Service Oriented Architecture*. Packt Publishing.
- [Papazoglou, 2003] Papazoglou, M. P. (2003). Service -Oriented Computing: Concepts, Characteristics and Directions. *Web Information Systems Engineering, International Conference on*, 0:3.
- [Papazoglou, 2008] Papazoglou, M. P. (2008). *Web Services: Principles and Technology*. Pearson, Prentice Hall.
- [Papazoglou and Heuvel, 2006] Papazoglou, M. P. and Heuvel, W. J. V. D. (2006). Service Oriented Design and Development Methodology. *Int. J. Web Eng. Technol.*, 2(4):412–442.

- [Parr, 2009] Parr, T. (2009). *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf.
- [Parra et al., 2009] Parra, C. A., Blanc, X., and Duchien, L. (2009). Context Awareness for Dynamic Service-Oriented Product Lines. In Muthig, D. and McGregor, J. D., editors, *SPLC*, volume 446 of *ACM International Conference Proceeding Series*, pages 131–140. ACM.
- [Pawlak et al., 2005] Pawlak, R., Duchien, L., and Seinturier, L. (2005). CompAr: Ensuring Safe Around Advice Composition. In Steffen, M. and Zavattaro, G., editors, *FMOODS*, volume 3535 of *Lecture Notes in Computer Science*, pages 163–178. Springer.
- [Peltz, 2003] Peltz, C. (2003). Web Services Orchestration and Choreography. *Computer*, 36(10):46–52.
- [Perrouin et al., 2008] Perrouin, G., Klein, J., Guelfi, N., and Jézéquel, J.-M. (2008). Reconciling Automation and Flexibility in Product Derivation. In *SPLC*, pages 339–348. IEEE Computer Society.
- [Pessemier et al., 2008] Pessemier, N., Seinturier, L., Duchien, L., and Coupaye, T. (2008). A Component-Based and Aspect-Oriented Model for Software Evolution. *International Journal of Computer Applications in Technology*, 31:94–105.
- [Popovici et al., 2002] Popovici, A., Gross, T., and Alonso, G. (2002). Dynamic Weaving for Aspect-Oriented Programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA. ACM.
- [Pourraz, 2007] Pourraz, F. (2007). *Diapason : une approche formelle et centrée architecture pour la composition évolutive de services Web*. PhD thesis, Université de Savoie.
- [Prehofer, 1997] Prehofer, C. (1997). Feature-Oriented Programming: A Fresh Look at Objects. In Aksit, M. and Matsuoka, S., editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Berlin / Heidelberg. 10.1007/BFb0053389.
- [Recker and Mendling, 2006] Recker, J. and Mendling, J. (2006). On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages.
- [Reddy et al., 2006] Reddy, Y., Ghosh, S., France, R., Straw, G., Bieman, J., McEachen, N., Song, E., and Georg, G. (2006). Directives for Composing Aspect-Oriented Design Class Models. In *Transactions on Aspect-Oriented Software Development I*, pages 75–105.
- [Rho et al., 2006] Rho, T., Kniesel, G., and Appeltauer, M. (2006). Fine-grained Generic Aspects. In Leavens, G., Clifton, C., LÃdmmel, R., and Mezini, M., editors, *Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), in conjunction with Fifth International Conference on Aspect-Oriented Software Development (AOSD.06), March 20-24, 2006, Bonn, Germany*. ACM.
- [Rose et al., 2008] Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. A. (2008). Constructing Models with the Human-Usable Textual Notation. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 249–263, Berlin, Heidelberg. Springer-Verlag.
- [Rosen, 2000] Rosen, K. H. (2000). *Handbook of Discrete and Combinatorial Mathematics, Second Edition*. CRC Press, 2 edition.
- [Roubtsova et al., 2009] Roubtsova, E. E., Wedemeijer, L., Lemmen, K., and McNeile, A. T. (2009). Modular Behaviour Modelling of Service Providing Business Processes. In Cordeiro, J. and Filipe, J., editors, *ICEIS (3)*, pages 338–341.
- [Russell et al., 2006a] Russell, N., Arthur, van der Aalst, W. M. P., and Mulyar, N. (2006a). Workflow Control-Flow Patterns: A Revised View. Technical report, BPMcenter.org.

- [Russell et al., 2006b] Russell, N., van der Aalst, W. M. P., ter Hofstede, A. H. M., and Wohed, P. (2006b). On the suitability of UML 2.0 activity diagrams for business process modelling. In *APCCM '06: Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling*, pages 95–104, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.
- [Sangiorgi and Walker, 2001] Sangiorgi, D. and Walker, D. (2001). *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA.
- [Sarkar and Brown, 1992] Sarkar, M. and Brown, M. H. (1992). Graphical Fisheye Views of Graphs. In *CHI'92: Proc. of the SIGCHI conf. on Human factors in computing sys.*, pages 83–91, New York, NY, USA. ACM.
- [Schärli et al., 2003] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A. (2003). Traits: Composable Units of Behaviour. In *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 327–339. Springer Berlin / Heidelberg. 10.1007/978-3-540-45070-2_12.
- [Schulte and Natis, 1996] Schulte, R. W. and Natis, Y. V. (1996). SSA Research Note SPA-401-068, Service Oriented Architectures, Part 1 & 2. Technical report, the Gartner Group.
- [Schwaber and Beedle, 2001] Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- [Seinturier et al., 2009] Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., and Stefani, J.-B. (2009). Reconfigurable SCA Applications with the FraSCaTi Platform. In *IEEE SCC*, pages 268–275. IEEE Computer Society.
- [Snyder, 1986] Snyder, A. (1986). Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, NY, USA. ACM.
- [Soa4All, 2009] Soa4All (2009). SOA4ALL Glossary. Technical report, SOA4ALL.
- [Sriplakich et al., 2006] Sriplakich, P., Blanc, X., and Gervais, M.-P. (2006). Supporting Collaborative Development in an Open MDA Environment. In *ICSM*, pages 244–253. IEEE Computer Society.
- [Stein et al., 2008] Stein, S., Barchewitz, K., and Kharbili, M. E. (2008). Enabling Business Experts to Discover Web Services for Business Process Automation. In Calisti, M., Walliser, M., Brantschen, S., Herbstritt, M., Gschwind, T., and Pautasso, C., editors, *Emerging Web Services Technology, Volume II*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 23–39. Birkhäuser Basel. 10.1007/978-3-7643-8864-5_3.
- [Stickel, 1981] Stickel, M. E. (1981). A Unification Algorithm for Associative-Commutative Functions. *J. ACM*, 28(3):423–434.
- [Stoerzer and Graf, 2005] Stoerzer, M. and Graf, J. (2005). Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 653–656, Washington, DC, USA. IEEE Computer Society.
- [Störrle, 2004] Störrle, H. (2004). Semantics and Verification of Data Flow in UML 2.0 Activities. In *Electronic Notes in Theoretical Computer Science. Elsevier Science Inc*, pages 35–52. Elsevier.
- [Stroustrup, 2000] Stroustrup, B. (2000). *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [Sun et al., 2009] Sun, Y., Demirezen, Z., Jouault, F., Tairas, R., and Gray, J. (2009). A Model Engineering Approach to Tool Interoperability. In Gasevic, D., Lämmel, R., and Van Wyk, E., editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 178–187. Springer Berlin / Heidelberg. 10.1007/978-3-642-00434-6_12.

- [Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [Taentzer, 2004] Taentzer, G. (2004). AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz, J. L., Nagl, M., and Böhlen, B., editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin / Heidelberg. 10.1007/978-3-540-25959-6_35.
- [Thaker et al., 2007] Thaker, S., Batory, D., Kitchin, D., and Cook, W. (2007). Safe Composition of Product Lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA. ACM.
- [Ungar and Smith, 1987] Ungar, D. and Smith, R. B. (1987). Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242.
- [Uzuncaova et al., 2010] Uzuncaova, E., Khurshid, S., and Batory, D. (2010). Incremental Test Generation for Software Product Lines. *IEEE Transactions on Software Engineering*, 36:309–322.
- [van der Aalst, 1998] van der Aalst, W. M. P. (1998). The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66.
- [van der Aalst et al., 2003] van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. (2003). Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51.
- [Vanderfesten et al., 2007] Vanderfesten, I., Cardoso, J., Mendling, J., Reijers, H. A., and Van Der Aalst, W. M. (2007). Quality Metrics for Business Process Models. *BPM and Workflow Handbook*, pages 179–190.
- [Verheecke et al., 2003] Verheecke, B., Cibrán, M., and Jonckers, V. (2003). AOP for Dynamic Configuration and Management of Web Services. In *Web Services - ICWS-Europe 2003*, volume 2853 of *Lecture Notes in Computer Science*, pages 55–85. Springer Berlin / Heidelberg. 10.1007/978-3-540-39872-1_12.
- [Vinoski, 2004] Vinoski, S. (2004). WS-Nonexistent Standards. *IEEE Internet Computing*, 8(6):94–96.
- [W3C, 2001] W3C (2001). Web Service Definition Language (WSDL). Technical report, W3C. <http://www.w3.org/TR/wsdl>.
- [W3C, 2004] W3C (2004). Web Service Glossary. Technical report, W3C.
- [W3C, 2005] W3C (2005). Web Services Choreography Description Language Version (WS-CDL) 1.0. Technical report, W3C. <http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/>.
- [W3C, 2007] W3C (2007). Simple Object Access Protocol (SOAP). Technical report, W3C. <http://www.w3.org/TR/2007/REC-soap12-part1-20070427/>.
- [W3C, 2008] W3C (2008). XML Schema Definition Language (XSD) 1.1. Technical report, W3C. <http://www.w3.org/TR/2008/WD-xmlschema11-1-20080620>.
- [Wang and Fung, 2004] Wang, G. and Fung, C. K. (2004). Architecture Paradigms and Their Influences and Impacts on Component-Based Software Systems. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90272.1, Washington, DC, USA. IEEE Computer Society.
- [White, 2005] White, S. A. (2005). Using BPMN to Model a BPEL Process. Technical report, BPTrends.
- [Whittle et al., 2009] Whittle, J., Jayaraman, P. K., Elkhodary, A. M., Moreira, A., and Araújo, J. (2009). MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In [Katz et al., 2009], pages 191–237.

- [Wimmer and Kramler, 2006] Wimmer, M. and Kramler, G. (2006). Bridging Grammarware and Modelware. In Bruel, J.-M., editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 159–168. Springer Berlin / Heidelberg. 10.1007/11663430_17.
- [Wohed et al., 2006] Wohed, P., Dumas, M., Hofstede, A. H. M. T., and Russell, N. (2006). On the Suitability of BPMN for Business Process Modelling. In *In Proceedings 4th International Conference on Business Process Management (BPM 2006), LNCS*, pages 161–176. Springer Verlag.
- [Wong and Hong, 2007] Wong, J. and Hong, J. I. (2007). Making Mashups With Marmite: Towards End-User Programming for the Web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1435–1444, New York, NY, USA. ACM Press.
- [Yellin and Strom, 1997] Yellin, D. M. and Strom, R. E. (1997). Protocol Specifications and Component Adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333.
- [Zhang et al., 2007] Zhang, J., Cottenier, T., van den Berg, A., and Gray, J. (2007). Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology*, 6(7):89–108.

Abstract

The Service Oriented Architecture (SOA) paradigm supports the assembly of atomic services to create applications that implement complex business processes. Since the design of a complete process can be very complex, composition mechanisms inspired by the *Separation of Concerns* paradigm (e.g., *features*, *aspects*) are needed to support the definition of large systems by composing smaller artifacts into a complex one. In this thesis, we propose ADORE, “an Activity meta-model supOrting oRchestration Evolution” to address this issue in the SOA context. The ADORE meta-model allows process designers to express in the same formalism business processes and *fragment* of processes. Such *fragments* define additional activities that aim to be integrated into others process. They can be composed into several processes and at different location through the use of algorithms which tame the complexity of large process design. Using these algorithms ensure properties in the final processes such as guard and activity order preservation. The underlying logical foundations of ADORE allow to define interference detection rules as logical predicate, such as consistency properties on ADORE models. Consequently, the ADORE framework supports process designers while designing large process, managing the detection of interference among fragments and ensuring that the composed processes are consistent and do not depend on the order of the composition. This work is illustrated in this document according to two case study: (i) JSEDUITE, an information broadcasting system daily used in several academic institutions and (ii) the CCCMS, a common case study to compare Aspect Oriented Modeling approaches. According to several collaboration with others research teams in various domain (i.e., requirement engineering, visualization and real-time systems), we expose as perspectives the integration of ADORE into a complete software development tool chain.

Résumé

Les Architectures Orientées Services (SOA) permettent la définition d’applications complexes par assemblage de service existants, par exemple sous la forme d’“orchestrations de services” implémentant des processus métiers. La complexité grandissante de ces assemblages impose l’utilisation de techniques telle que la Séparation des Préoccupations (e.g., *aspects*, *fonctionnalités*) pour en maîtriser la difficulté. Dans cette thèse, nous présentons ADORE, un métamodèle d’activité permettant l’évolution des orchestrations. Ce métamodèle permet aux *designers* d’orchestration d’exprimer dans le même formalisme (celui des processus métier) des “orchestrations” et des “fragments d’orchestrations”. Ces fragments définissent des activités additionnelles qui visent à être intégrée dans d’autres processus métiers. Nous proposons alors différents algorithmes de compositions permettant l’intégration automatique de ces fragments dans des processus existants. Ces algorithmes définissent un ensemble de propriétés de compositions (e.g., préservation des relations d’ordre, des conditions), et assurent leur respect dans les processus composés. De plus, les algorithmes définis dans ce cadre assurent que les résultats de composition obtenus ne dépendent pas de l’ordre d’application des fragments. Les fondations logiques sous-jacente à ADORE permettent la définition de règles de détection d’interférences dans les processus, sous la forme de prédicats logiques. Le canevas logiciel développé dans cette thèse propose ainsi un support aux *designers*, en identifiant les interférences entre fragments apparaissant lors des compositions. Nous illustrons cette contribution au travers de deux études de cas: (i) JSEDUITE, une application de diffusion d’information utilisée en production dans plusieurs établissement scolaires et (ii) CCCMS, une application de gestion de crise (accidents de voitures) implémentée dans le cadre d’une réponse à une étude de cas commune sur la modélisation orientée aspects (publiée dans le journal TAOSD). Pour conclure, nous mettons en perspectives de récentes collaborations avec des équipes à l’international, visant l’intégration d’ADORE au sein d’un processus de développement logiciel complet, allant des l’ingénierie des besoins à la production de code, en passant par la visualisation efficace des processus composés.