UNIVERSIDAD DE LOS ANDES
FACULTY OF ENGINEERING
Department of Systems and
Computer Engineering
Software Construction Group

VRIJE UNIVERSITEIT BRUSSEL
FACULTY OF SCIENCE
Department of Computer Science
Software Languages Lab

Executable Models for Extensible Workflow Engines

Mario Sánchez Puccini

A dissertation submitted in partial fulfilment of the requirements for the degrees of Doctor of Science at Vrije Universiteit Brussel and Doctor of Engineering at Universidad de los Andes

February, 2011

Promoters: Prof. Dr. Jorge Villalobos,
Prof. Dr. Viviane Jonckers,
Dr. Dirk Deridder

Abstract

In recent years, workflows have started to be used in many domains such as business processes, scientific applications, and e-learning. Accordingly, various workflow specification languages have needed to be designed in each one of those domains. Corresponding infrastructures (like editors, engines, and monitoring applications) to utilize and enact those languages have been implemented as well. Among the benefits commonly associated with workflows, one that stands out is the possibility of modifying workflow definitions by manipulating only high level or domain specific concepts. This is extremely positive because it enables domain experts to introduce changes to the systems without requiring the intervention of software developers. However, this flexibility is not always enough. Workflow users often encounter new requirements that cannot be properly solved with existing tools and languages because they depend on new concepts or should use different structures. In those cases, their only options are to change or extend the workflow languages, or to develop entire new ones. Unfortunately, nowadays there is poor support to implement either alternative. With the former, the biggest problem is that workflow languages and their associated tools have very limited extensibility capabilities. With the latter, there are two different problems. The first one is the scarcity of frameworks or libraries available to support the development of new workflow engines to enact the newly created languages. The second problem is that existing engines are tightly coupled to the languages they were developed for. Therefore, by creating a new language one looses existing tool support (editors, simulators, monitoring applications, and others).

The goal of this dissertation is to solve these limitations by offering a platform that serves as the foundation for extensible workflow engines. In this way, the enactment of new workflow languages will be more easily supported, because the implementation of every engine will not start from scratch. Furthermore, this platform is geared towards supporting extensible and flexible workflow languages, and thus changing requirements will be more easily accommodated. The proposed platform supports various kinds of workflow languages. In the first place, there are general purpose workflow languages that can be used in many domains, such as BPEL or BPMN. Another kind is that of domain specific workflow languages, such as IMS-LD or Sedna. Finally, the proposed platform also supports concern specific workflow languages, which modularize workflow descriptions in accordance with various possible criteria.

There are three ideas that are central to the proposal. First of all, there is the idea of using metamodels to define the structure of workflow languages, and using models to represent specific workflows. The second idea is that of making the models executable by establishing executable semantics for every element in the metamodels, and following the semantics of the language. Finally, the third idea is that of coordinating the execution of several executable models in order to support the aforementioned concern specific languages. To support these three ideas we developed the notion of 'open objects', which are used to define the behavior of elements of the metamodels in a special way. By using open objects, each element defines its own semantics. However, the actual coordination between those elements is specified in an explicit and flexible way. It is the responsibility of a special kernel in the platform to use this information to coordinate the behavior of the elements and thus execute the workflows. Furthermore, this kernel also provides other features common to workflow systems. Therefore, these features do not have to be reimplemented for every language.

The ideas presented in this dissertation have been implemented in the Cumbia platform, which encompasses a Java based development framework, and two components called Cumbia Kernel and Cumbia Weaver. Concretely, the framework is what workflow language developers need to build their own metamodels based on open objects. The Cumbia Kernel is the component that loads and runs the models. The Cumbia Weaver is what establishes relationships between models conformant to different metamodels, thus allowing the interaction of concern specific languages.

This approach has been validated with the construction of engines for well known workflow languages, which include BPEL, BPMN and YAWL. Other experiments include the implementation of engines for concern specific workflow languages (miniBPMN, XPM, and XTM), and for domain specific workflow languages (IMS-LD and PaperXpress).

Samenvatting

Workflows worden de laatste jaren steeds meer gebruikt in tal van domeinen zoals business processen, wetenschappelijke toepassingen en e-learning. Ter ondersteuning werd voor al deze domeinen een workflow specificiatietaal ontwikkeld. Tevens werden ook overeenkomstige infrastructuren (zoals workflow editors en engines) geïmplementeerd. Een opvallend voordeel van workflows is de mogelijkheid om workflow definities aan te passen en te manipuleren op een hoog niveau of op domeinspecifieke concepten. Hierdoor kunnen domeinexperts het systeem zelf aanpassen, zonder hulp van softwareontwikkelaars. Zelfs deze flexibiliteit volstaat niet altijd aangezien workflowgebruikers vaak nieuwe vereisten tegen komen die niet kunnen worden opgelost met behulp van bestaande tools en talen daar ze stoten op nieuwe concepten of andere structuren gebruiken. Indien dit het geval is, kunnen ze de workflowtalen nog proberen aanpassen of uitbreiden of een nieuwe taal van nul af aan ontwikkelen. Helaas biedt men tegenwoordig weinig ondersteuning om een van deze opties te implementeren. De grootste problemen omtrent het eerste alternatief zijn de beperkte uitbreidingsmogelijkheden van workflowtalen en bijhorende tools. Bij het laatste alternatief staat men voor twee problemen. Ten eerste is er een tekort aan beschikbare frameworks of bibliotheken om de ontwikkeling van nieuwe workflow engines te ondersteunen. Ten tweede zijn bestaande engines te nauw verbonden met de workflowtalen waarvoor ze werden ontwikkeld. Bestaande ondersteuning gaat hierdoor verloren wanneer een nieuwe taal wordt ontwikkeld.

Het doel van deze thesis is tot een oplossing te komen voor de reeds vermelde beperkingen door een platform aan te bieden dat als basis dient voor uitbreidbare workflow engines. Op deze manier zal het uitbrengen van nieuwe workflowtalen gemakkelijker worden ondersteund aangezien men bij de implementatie van de engines niet telkens van nul af aan moet herbeginnen. Bovendien is dit platform gericht op de ondersteuning van uitbreidbare en flexibele workflowtalen waardoor het eveneens gemakkelijker wordt vereisten te wijzigen. Verschillende soorten workflowtalen worden ondersteund door het voorgestelde platform. In de eerste plaats ondersteunt het platform algemene workflowtalen zoals BPEL en BPMN die in verscheidene domeinen worden gebruikt. Een ander soort talen die ondersteund worden zijn domeinspecifieke workflowtalen zoals IMS-LD en Sedna. Tenslotte ondersteunt het voorgestelde platform eveneens concern specifieke workflowtalen die workflow beschrijvingen modelleren in overeenstemming met verschillende mogelijke criteria.

Drie ideeën staan centraal in dit voorstel. Eerst en vooral het gebruik van metamodellen om de structuur van workflowtaal te preciseren en modellen te gebruiken om specifieke workflows voor te stellen. Ten tweede maakt men modellen uitvoerbaar door een uitvoerbare semantiek voor elk element in deze metamodellen te voorzien en door de semantiek van de taal te volgen. Het derde idee staat voor het coördineren van de uitvoering van workflow over verschillende

uitvoerbare modellen om ondersteuning te bieden aan de reeds vermelde concern specifieke workflowtalen. Ter ondersteuning van deze drie ideeën introduceerden we de notie ‘open objecten’ die worden gebruikt om het gedrag van de elementen van metamodellen op een speciale manier te definiëren. Door open objecten te gebruiken, definieert elk element zijn eigen semantiek. De eigenlijke coördinatie tussen deze elementen is echter gespecificeerd op een expliciete maar flexibele manier. Het is immers de verantwoordelijkheid van een speciaal ontwikkelde kernel in het platform om deze informatie te gebruiken om het gedrag van deze elementen te coördineren en dus in te staan voor de uitvoering van de workflows. Bovendien voorziet deze kernel ook andere features eigen aan workflow systemen, waardoor deze eigenschappen niet voor elke taal opnieuw moeten worden geïmplementeerd.

De ideeën die in deze thesis worden voorgesteld, werden geïmplementeerd in het Cumbia platform. Dit platform omvat een Java ontwikkelingsplatform en tevens ook twee componenten genaamd de Cumbia kernel en de Cumbia weaver. Door gebruik te maken van het framework zijn de ontwikkelaars van workflowtalen in staat hun eigen metamodellen te creëren aan de hand van open objecten. De Cumbia kernel is de component die verantwoordelijk is voor het inladen en uitvoeren van de modellen. De Cumbia weaver daarentegen initieert de relaties tussen de verschillende modellen in overeenstemming met de verscheidene metamodellen, waardoor de interactie van concern specifieke taken mogelijk wordt. Deze benadering werd gevalideerd door de ontwikkeling van workflow engines voor bekende workflowtalen, waaronder BPEL, BPMN en YAWL. Andere experimenten omvatten het implementeren van engines voor concern specifieke workflowtalen (miniBPMN, XPM, en XTM) en voor domeinspecifieke workflowtalen (IMS-LD and PaperXpress).

Acknowledgements

Getting a Ph.D. is the most challenging thing I've ever attempted: for more than four years, I've committed the best of my energies to this piece of work, and it's almost incredible that it fits in under 300 pages. Producing it has been tough and I've learned a lot in the process: I've started to develop many important abilities that I previously lacked. But more importantly, I've experienced first-hand the commitment required to attempt really difficult things. This has also been an opportunity to see how supportive are my colleagues, friends, and family members. I'm happy for having realized how important they all are for my life: only my name is in the cover of this dissertation, but many others rightfully deserve a share of the compliments.

I have been working under the guide of Prof. Dr. Jorge Villalobos since I started my masters degree and I can only have words of admiration, appreciation and gratitude for everything he has given and taught me. He has always appreciated my opinions, even in the frequent cases where I've been wrong. Our discussions have always been constructive and at the end of each one I've learned something. I would like to thank him for the confidence he put on me and for the time he committed to guiding my work. Dr. Dirk Deridder was my promotor at the VUB, and since I arrived in Belgium he started questioning my ideas from angles I had never considered before. He also made me question assumptions that I never analyzed in detail before. This was very valuable, and I learned continuously from it. I would also like to thank Dirk for his time, his effort, and his willingness to start working with three guys from the other side of the ocean, in topics that were not among his top priorities. Prof. Dr. Viviane Jonckers was also my promotor, and I regret that the conditions of my Ph.D. made it difficult to work more closely to her. However, the feedback I got from her always was extremely valuable. In particular, I would like to thank her for her efforts to translate my words into SSEL-PROG-SOFT terms, for analyzing my ideas from different points of view that I was not initially considering, and for always trying to find the value of the ideas I presented. At some points of this whole process, I thought that having more than one promotor was too complex and risky. Now I see how this was a great opportunity, as each one of them taught me different things and helped me to develop different abilities.

Many thanks also go to Prof. Dr. Rubby Casallas, which has given me a lot of support since the inception of the Caramelos project. In many opportunities she gave me feedback on my work, but also opportune support, words of encouragement, and advices.

I would also like to thank the members of my Ph.D. committee, for the time they put to read this dissertation and for the valuable feedback that they gave me. This committee was composed by Jorge, Dirk, Vivianne, and Rubby, and also by Prof. Dr. Wolfgang De Meuter, Prof. Dr. Beat Signer, Prof. Dr. Jean-Claude Royer, Prof. Dr. Jacky Estublier, and Prof. Dr. Darío Correal.

While doing all the work related to this dissertation, I had the opportunity of being a part of SSEL and SOFT, at the VUB. I'm very thankful to all the members of these labs, and especially to Theo, Wolf, and Viviane for allowing me to participate in an environment that is challenging, diverse, and always interesting. Among the numerous members of these labs, I would like to single out those that had a greater impact on my work: Niels, Ragnhild, Dennis, Andy, and Eline. Eline also gave me extremely valuable feedback at key points of the writing process, and I hope I'll be able to pay her back when she'll start writing her dissertation. A big thank you also to all the good friends I made in Belgium and made me feel welcomed at all time: Isabel, Nicolás, Rob, Carlos, Angela, Tania, Jorge, Sonia, Sebastián, Elisa, Coen,Stefan, Eline, Frank, El Mono and Nani.

Andrés Yie, Oscar González and Adriana Manrique, deserve a big thank you for their participation in this process. Four years ago they were barely known colleagues. After the Ph.D., and especially after the Belgium experience, I'm happy I've met them and I'm happy to call them Friends. Andrés, Oscar and Adriana: I've learned a lot from you, I've enjoyed your company, and I'm thankful for your support and friendship.

I also need to thank those at Uniandes that have been related to Cumbia and have had a part on its development and validation. At the risk of forgetting someone (please forgive me if I do), I'll attempt to name them all: Pablo B., Manuel, Camilo, Nadya, Carlos V., Pablo M., John E., Daniel R., Daniel A., Iván, Sergio, Nolberto, Gilberto, Alex, John C., Mauricio, Jimmy, Silvia, Carlos R. and Darío. I'm especially thankful at those of you that are still around and have offered me their friendship. Many thanks also go to Felipe, Rafael and Juan Erasmo, for being around and willing to engage in interesting discussions.

I've been blessed with a happy and very supportive family, from my parents and siblings, to my grandmothers, uncles and aunts. Especially in my closest family, everyone has always shown interest in the things I've done, and it's been great to have fans at home. María Beatriz and Paolo: I'm sorry for all the technical discussions you've endured over dinner and I admire your stoicism. Papá y Mamá, for many reasons I admire you both and I've tried hard to follow your lead. I hope I've met your expectations. Thank you for your guidance, for your love, and for always keeping your faith on me.

And at last, I have to thank you, Gigi, for your unconditional love, and your constant support that always comes with a smile. During all these years, I've learned a lot from you, and I hope one day I'll be as generous and kind as you are with everyone around you. I don't know if I'll ever be able to pay back this big debt I now have, but I'll try hard for the rest of my life.

This dissertation, and the dissertations of Andrés Yie and Oscar González have been developed within the frame of Caramelos, a VLIR funded project. It is reasonable to wonder why an institution of the Flemish government, which has its offices at more than 8500 kms from Bogotá, decided to invest in the education of three graduate students at the Universidad de los Andes. The rationale for this can be found in the Caramelos project; on the commitment of Belgium towards the development of science and research; and on the willingness of Belgians to share opportunities with less fortunate countries. Without this support, I would had never started doing research. Therefore, I offer a lot of thanks to VLIR, and to those that participated in the elaboration of the proposal for the project:

Rubby Casallas, Dirk Deridder, Sebastián González, Theo D'Hondt, and Linda Dasseville. I hope that I will live up to the expectations that you had more than 5 years ago, when you first presented the project.

Finally, to all of you that were mentioned, and to those that I forgot and will rightfully complain, thanks a lot.

Table of Contents

Abstract i

Samenvatting iii

Acknowledgements v

Table of Contents ix

List of Figures xiii

List of Listings xvii

List of Tables xix

1 Introduction **1**
- 1.1 Research context . . . 1
- 1.2 Problem statement . . . 4
  - 1.2.1 Challenges to the development, extension, and adaptation of workflow engines . . . 4
  - 1.2.2 Objective . . . 8
- 1.3 Thesis approach . . . 10
- 1.4 Thesis contributions . . . 15
- 1.5 Organization of the dissertation . . . 16

2 Workflow Modeling and Enactment **19**
- 2.1 General workflow concepts . . . 19
- 2.2 Workflow modeling . . . 23
  - 2.2.1 Workflow specification languages . . . 23
  - 2.2.2 Execution models . . . 32
  - 2.2.3 Expressive power and suitability . . . 36
- 2.3 Workflow modularization . . . 38
  - 2.3.1 Business process perspectives . . . 39
  - 2.3.2 Aspect orientation in workflow systems . . . 41
- 2.4 Workflow kernels and intermediate languages . . . 42
  - 2.4.1 Intermediate workflow languages and models . . . 43
  - 2.4.2 Database based . . . 46
  - 2.4.3 Object oriented kernels . . . 48
  - 2.4.4 ECA Rules . . . 50
- 2.5 Development of engines for new languages . . . 51

3 Executable Models in a Workflow Kernel **55**
- 3.1 A metamodeling platform for executable workflow models . . . 55

3.2 The role of a configurable kernel . . . 59
3.3 Concern specific workflow languages . . . 61
3.4 Coordination of concern specific models . . . 63
3.5 Open Objects . . . 64
3.6 The Cumbia platform . . . 68
3.7 Summary . . . 69

4 Workflow Models based on Open Objects 71
4.1 A metamodeling platform based on Open Objects . . . 72
4.1.1 Metamodeling platforms . . . 72
4.1.2 Model behavior . . . 73
4.1.3 The role of open objects . . . 74
4.2 Open objects . . . 74
4.2.1 General characteristics . . . 74
4.2.2 Open objects interaction . . . 78
4.2.3 Navigation language . . . 84
4.2.4 Open object definition . . . 90
4.2.5 Extension mechanisms . . . 95
4.2.6 Other perspectives on open objects . . . 104
4.3 Metamodels and model definition . . . 106
4.3.1 Metamodel definition . . . 107
4.3.2 Model definition . . . 107
4.4 Model execution . . . 113
4.4.1 Model Instantiation . . . 114
4.4.2 Model Execution . . . 115
4.5 Workflow engines and the Cumbia Kernel . . . 117
4.6 A sample metamodel: `MiniBPMN` . . . 120
4.6.1 Structure of `MiniBPMN` . . . 120
4.6.2 Process and Task . . . 122
4.6.3 Roles in `MiniBPMN` . . . 123
4.6.4 An engine for `MiniBPMN` . . . 124
4.7 A sample `MiniBPMN` model . . . 126
4.8 Extensions to the `MiniBPMN` . . . 127
4.8.1 Extension 1: AdHocBlock . . . 128
4.8.2 Extension 2: AdHocTask . . . 130
4.8.3 Extension 3: Unsupervised Process . . . 130
4.8.4 Extension 4: Execution Logger . . . 132
4.9 Summary . . . 133

5 Coordination of Multiple Concern Specific Models 137
5.1 Model composition and coordination . . . 138
5.1.1 Model composition techniques . . . 138
5.1.2 Cumbia model composition strategy . . . 141
5.1.3 A note about the graphical syntax . . . 142
5.2 CCL – The Cumbia Composition Language . . . 144
5.2.1 Structure of a CCL program . . . 145
5.2.2 Instructions to create model instances . . . 147
5.2.3 Instructions to create links . . . 147
5.3 An architecture to execute concern specific models . . . 148
5.3.1 General architecture . . . 149

5.3.2 The Cumbia Weaver 151
5.4 A sample workflow 153
5.4.1 The metamodels 153
5.4.2 The control model 156
5.4.3 The time models 156
5.4.4 The composition code 158
5.4.5 Packaging and using the assembly 163
5.5 Summary 164

6 Towards a development process for workflow engines based on Cumbia 167
6.1 Process overview 167
6.2 Design and develop languages, metamodels and engines 171
6.2.1 Design a language 172
6.2.2 Metamodel design and implementation 173
6.2.3 Metamodel testing 175
6.2.4 Engine development 181
6.2.5 Language and metamodel evolution 181
6.3 Design metamodel composition 183
6.3.1 Select languages 184
6.3.2 Analyze language interactions 184
6.3.3 Design metamodel composition guidelines 185
6.3.4 Test metamodel composition 186
6.4 Design and build applications 186
6.5 Design and build workflows 188
6.5.1 Design a workflow 188
6.5.2 Build models, extensions, and resources 188
6.5.3 Write CCL programs and package the assemblies 190
6.6 Enact workflows 190
6.7 Summary 192

7 Validation 193
7.1 Objective revisited 193
7.2 Overview of the case studies 195
7.3 `MiniBPMN` 197
7.4 YAWL and Petri nets 198
7.5 PaperXpress 207
7.6 IMS-LD 214
7.7 Other experiments 220
7.7.1 XPM 220
7.7.2 BPMN 222
7.7.3 WS-BPEL 225
7.7.4 SCA 226
7.8 Review of the findings 229

8 Conclusion 233
8.1 Summary and reflection 233
8.2 Conclusions 236
8.3 Future work 238
8.3.1 Possible improvements and additions for Cumbia 238

8.3.2 Concrete syntax 241
8.3.3 Management of evolution 241

A Interfaces of the Open Objects 243
A.1 INavigable 243
A.2 IModelElement 248
A.3 IOOKernelElement 248
A.4 IOpenObject 250

B Metamodel and Model Specification Schemas 251
B.1 Metamodel definition schema 251
B.2 State machine definition schema 256
B.3 Model definition schema 257

C Navigation Language 261

D Cumbia Composition Language - CCL 265

Bibliography 269

List of Figures

1.1 Basic workflow concepts 2
1.2 Metamodeling and workflows 11
1.3 An execution platform based on metamodels 11
1.4 Concern specific workflow languages and definitions 13
1.5 Coordination of workflow concerns 14

2.1 Components and interfaces defined in the Workflow Reference Model [Hol95] 22
2.2 Sample BPMN diagram 25
2.3 A sample YAWL process. The net on the upper part is the top level net, while the one on the lower part of the image is the sub-net. 28
2.4 Conceptual model of IMS-LD [IMS03b] 31
2.5 A sample BPMN and two possible mappings to Petri nets 34
2.6 A simple BPMN process 36
2.7 The core of the AMFIBIA model [KAR06] 41
2.8 Actions and Events associated to places and transitions [FF04] . 44
2.9 Extension to the functionalities of the workflow engine [FF04] . . 44
2.10 System architecture of OPERA [AHST97b] 47
2.11 The different language representations in OPERA. Adapted from [AHST97b] 48
2.12 Workflow metamodel implemented in Lewfe [Pér09] 50

3.1 A metamodeling platform and the workflow context 57
3.2 Sample BPMN process with control, resources, time, and data. . 62
3.3 Coordination of concern specific models. 64
3.4 Basic components of an open object 65
3.5 Main elements of the Cumbia Platform 69

4.1 Basic notation to represent open objects 75
4.2 Entity for the open object `Task` 75
4.3 State machine for the open object `Task` 77
4.4 Events generated when a transition is triggered 79
4.5 Simplfied BPMN metamodel 80
4.6 Sample BPMN process 81
4.7 Coordination of the entity and the state machine of `Task` 82
4.8 Interaction between `Task` and `Gateway` 82
4.9 State machines of *a*) `Process` and *b*) `End Event` 83
4.10 Differences between *a*) method invocation and *b*) event passing . 85
4.11 Basic classes and interfaces in the open object's framework 95
4.12 Hierarchy of extensions 96

4.13 a) The configuration approach (BPEL) vs. b) the specialization approach (YAWL). 99
4.14 Inconsistency scenario 100
4.15 New intermediate state possibilities: located a) before, and b) after 103
4.16 Life cycle of a `Work Item` in YAWL [The08] 104
4.17 A model instance at run time 113
4.18 Events generated when a transition is triggered 116
4.19 Basic architecture of a Cumbia based engine 118
4.20 Asynchronous messages in a BPEL engine 119
4.21 `MiniBPMN` sample process with legend 120
4.22 Structure of `MiniBPMN` 121
4.23 State machine of `BasicElement` 122
4.24 Open object for `Process` in `MiniBPMN` 122
4.25 Structure of `Task` in `MiniBPMN` 123
4.26 Architecture of the engine for `MiniBPMN` 125
4.27 `MiniBPMN` sample process 126
4.28 Objects diagram corresponding to the sample process 127
4.29 Extended `MiniBPMN` metamodel 128
4.30 Extended `MiniBPMN` metamodel 128
4.31 Extended element: `AdHocBlock` 129
4.32 Extended element: `AdHocTask` 130
4.33 New element: `ExecutionLogger` 132

5.1 Graphical syntax example 143
5.2 Architecture of a workflow solution based on Cumbia 149
5.3 Representation of the run time state of the Cumbia Weaver and of two engines 152
5.4 `MiniBPMN` metamodel 154
5.5 Fragment of the `XTM` metamodel 155
5.6 Relations between the metamodels in the sample workflow 155
5.7 The control concern in the sample workflow 156
5.8 Integration between control and time in the sample workflow . . 157
5.9 Objects diagram of the first time restriction 157
5.10 Objects diagram of the second time restriction 158
5.11 Objects diagram of the third time restriction 158
5.12 Partial view of the `MiniBPMN` model composed with the first `XTM` model 161
5.13 Partial view of the `MiniBPMN` model composed with the second `XTM` model 162
5.14 Partial view of the `MiniBPMN` model composed with the third `XTM` model 163
5.15 Structure of the *cumbiar* file with the sample assembly. 164

6.1 The five main phases in the process for developing workflow languages and engines based on Cumbia 168
6.2 Graphical notation used to present the details about the process 170
6.3 Design and implementation of a Cumbia based workflow language 172
6.4 The OO Editor editing a metamodel structure, and an open object 175
6.5 Scenario based testing of metamodels 176
6.6 Screenshot of the Cumbia Debugger 177

6.7 The phases of the execution of a scenario 180
6.8 Steps to evolve an existing workflow language 182
6.9 Details of the process to design the composition of metamodels . 183
6.10 Details of the process to design and build applications 187
6.11 Details of the process to design and build workflows 189
6.12 Details of the process to enact workflows 191

7.1 Characteristics of the case studies 196
7.2 The 5 dimensions of a process in YAWL 199
7.3 Symbols used in YAWL [vdAtH06] 200
7.4 Concern specific metamodels in YOC 202
7.5 The control metamodel for YOC (divided in parts for readability) 203
7.6 Shared condition in a YAWL net 204
7.7 Interlocked shared conditions in a YAWL net 204
7.8 The Petri nets client showing one of the problematic cases 205
7.9 Control and time metamodels in YOC 206
7.10 The structure of an article in PaperXpress 208
7.11 A part of the process to write and review an article using PaperXpress 208
7.12 Basic metamodels in PaperXPress 209
7.13 `PxControl`: metamodel for the control concern in PaperXpress . 209
7.14 `PxArticle`: metamodel for the article concern in PaperXpress . . 209
7.15 The metamodels used in PaperXPress and their extensions 210
7.16 New actions in the state machines of a) `PxActivity` and b) `Section`211
7.17 New elements in the `Extended PxArticle` metamodel 212
7.18 Elements in the `PxAuthors` metamodel: a) `Assignment` and b) `Author` 213
7.19 Metamodels in Garabato 215
7.20 Elements in the `CumbiaLD` metamodel 215
7.21 The XLD metamodel to support levels B and C of IMS-LD . . . 217
7.22 The `RoleD` metamodel 217
7.23 The `LDContents` metamodel 217
7.24 Some screenshots from Garabato 218
7.25 The main elements of the `DMM` metmaodel 219
7.26 Metamodel of `XPM` 220
7.27 A process modeled with `XPM` 221
7.28 Control flow routing in `XPM` 222
7.29 Metamodels in Alegre 223
7.30 Mapping of a BPMN XOR-Split into `XPM_BPMN` 223
7.31 Alegre structure and run time behavior 224
7.32 Schematic view of the WS-BPEL Engine (Caffeine) 225
7.33 Elements of SCA, based on figures from [OSO07] 227
7.34 A sample SCA application. This figure can be found in [Bar07] . 228
7.35 Metamodel for SCA (see note) 228

B.1 Graphical representation of the metamodel definition schema . . 252
B.2 Graphical representation of the state machine definition schema . 256
B.3 Graphical representation of the model definition schema 258

List of Listings

4.1 The implementation of the entity of `Task` 75
4.2 Generation of an event in an entity 78
4.3 Extract of the method `setToken()` in the entity of `Task` 81
4.4 Implementation of action *giveToken* 83
4.5 Java code equivalent to the recursive navigation expression . . . 89
4.6 Structure of an open object definition 90
4.7 Structure of a state machine definition 92
4.8 Extract of the implementation of the entity of an open object . . 93
4.9 Implementation of an action 95
4.10 Extension example: new interface 97
4.11 Extension example: additional events 97
4.12 Extension example: additional roles 97
4.13 Extension example: new relations 97
4.14 Extension example: new implementation of the entity. 98
4.15 Empty state machine extension 100
4.16 Extension example: additional action 100
4.17 Extension example: additional transition 101
4.18 Extension example: new state 102
4.19 Extension example: new intermediate state 102
4.20 Extension example: new state machine. 103
4.21 Basic structure of a metamodel definition 107
4.22 Basic structure of a model definition 108
4.23 Definition of model-specific metamodel 110
4.24 Runtime controller configuration 110
4.25 Definition of the elements of a model 110
4.26 Definition of connections between elements of a model 111
4.27 Definition of the `AdHocBlock` extension (without the state machine) 129
4.28 Definition of the `AdHocTask` extension 130
4.29 Definition of the `UnsupervisedProcess` extension 132
4.30 Implementation of the new action NotifyProcessEnded 132
4.31 Definition of the `ExecutionLogger` extension 133
5.1 Basic elements of a CCL program 145
5.2 Block of instructions executed when the assembly is instantiated 146
5.3 Blocks of instructions executed when model instances are created 146
5.4 CCL Example: Model instantiation 147
5.5 CCL Example: Action Link 148
5.6 CCL Example: Event based Link 148
5.7 Assembly declaration and initialization of the assembly instance 159
5.8 Composition of the first time model 159
5.9 Composition of the second time model 160
5.10 Composition of the third time model 160

7.1 Definition of the `PersistentSection` extension 211
A.1 INavigable 243
A.2 IModelElement 248
A.3 IOOKernelElement 248
A.4 IOpenObject 250
B.1 Metamodel definition schema 251
B.2 State machine definition schema 256
B.3 Model definition schema 257
D.1 CCL Xtext specification 265

List of Tables

2.1 Representation as ECA rules of the process in figure 2.6 36

4.1 Some roles evaluated in the sample process 125

5.1 Types of lines in the proposed graphical syntax 143

7.1 Match between case studies and characteristics of the platform . 230

C.1 Tokens 262
C.2 NON-Terminals 263

1 Introduction

1.1 Research context

Reduced to its core elements, workflow technology serves to control and coordinate, in a flexible way, the execution of sets of well defined operations. This technology has recently gained momentum in many different domains, including business processes, scientific applications, and e-learning, and one reason for this, is that workflows abstract away from technical concepts by employing concepts closer to each specific domain. Because of this, workflows are often used by domain experts which do not have a solid technical background. Furthermore, workflow based applications are flexible and can be easily changed by deploying new workflow definitions. Thus, workflows are often used in contexts where requirements change frequently [GHS95]. As an example, consider that the profitability of many companies depends on having processes that manage efficiently the requirements of the market and the pressures from competitors. By using workflows to represent and support the enactment of those business processes, companies reduce the time and effort required to adapt their systems to updated processes.

A *workflow management system* (WfMS) is a package that supports the definition, enactment, and control of *workflow definitions*, while allowing interactions of human participants and external applications [Coa99, GV01, GHS95]. A WfMS usually includes editors to create the definitions, management applications to control the execution, client applications to allow the interaction of participants, and monitoring applications which track the execution of the workflows and are used to derive and register relevant metrics. The central element in a WfMS is a *workflow engine*, which loads and instantiates workflow definitions, and enacts them according to the semantics of the workflow specification language in use. Figure 1.1 depicts the relationship between the different workflow concepts.

Nowadays, a large number of workflow specification languages have been designed and implemented. Using those, it is possible to describe workflows in a large number of contexts and with relatively high level concepts. Compared to general purpose programming languages, workflow specification languages work

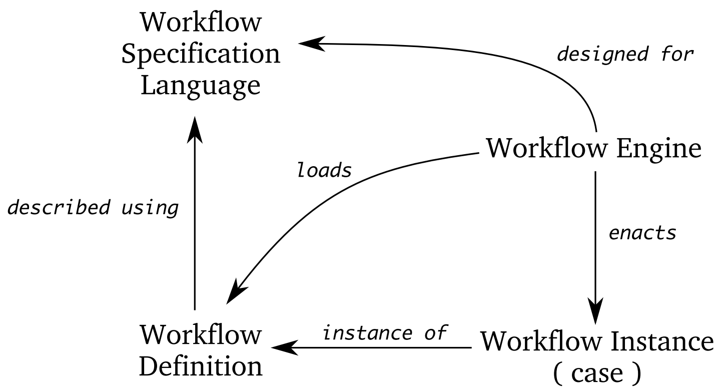


Figure 1.1 Basic workflow concepts

on a less technical level of abstraction, and are only useful for restricted kinds of problems. In general, different workflow languages tend to manage different concepts depending on the domain they target. However, there is also a lot of overlap between some of those languages; in many cases, the differences between them are minimal.

Although a formal classification does not yet exist, workflow specification languages can be classified as generic workflow languages or domain specific workflow languages. *Generic workflow languages* (GWfLs) are those languages that do not target a particular domain, and thus can be used in a large number of contexts. In this sense, they are similar to generic programming languages[1] [vDKV00]. One of the best known GWfLs is WS-BPEL (or BPEL) [OAS05]: although it is originally a language for defining web-services' composition, it has been used to describe workflows in many different domains, including business processes, web applications, scientific applications, and computer-aided engineering. Other well-known examples of GWfLs include BPMN [Obj08, Obj09a], and XPDL [Coa08]. These three languages offer different approaches to business process modeling, and they have many supporters and compatible applications. Currently, there are at least 9 BPEL engines, 61 implementations of BPMN reported to the OMG [Obj09b], and 83 tools or companies that support XPDL [Coa09]. This has happened mainly because these languages have been standardized (BPEL by OASIS[2], BPMN by the OMG[3], and XPDL by the WfMC[4]), and because they have been proposed and adopted by influencing companies such as IBM, Oracle and Microsoft.

In the opposite end of the spectrum are *domain specific workflow languages*

[1]In [vDKV00], van Deursen, Klint and Visser discuss the dichotomy between *domain specific* languages and *generic* programming languages. The focus of this dissertation is on workflow languages which are not *general purpose*, but can be either *generic* or *domain specific*.

[2]The Organization for the Advancement of Structured Information Standards (OASIS) is a consortium that drives the development, convergence and adoption of open standards. The standards maintained by OASIS include XML and several related to web-services.

[3]The Object Management Group (OMG) is an international consortium that develops technology standards. The members of the OMG include both companies that do software development and end-users that provide feedback and real usage scenarios. Some of OMG's standards include UML and CORBA.

[4]The Workflow Management Coalition (WfMC) is an organization formed by companies and institutions that work with workflows and especially with BPM. The standards proposed by the WfMC have as main goal to enable interoperability between tools developed by different vendors.

(DSWfLs). Similarly to a Domain Specific Language (DSL), a domain specific workflow language focuses on a particular workflow domain and offers a restricted suite of concepts and constructs [vDKV00]. Because of this, the elements included in a DSWfL are better aligned to domain concepts with respect to any GPWfL. An example of a DSWfL is Sedna [WEB$^+$07], a language for grid-based applications that was derived from BPEL. In addition to the *generic* elements offered by BPEL, Sedna also includes elements needed in this domain such as *concurrent experiments*, *indexed flows*, and *hierarchical decompositions*. IMS-LD [IMS03b] is another example of a DSWfL. This language targets specifically the e-learning domain, and it includes domain concepts such as *learning sequence*, *learning material*, and *learner*.

The characteristics of workflows make them suitable for contexts where requirements change frequently. In many cases, these changes can be handled by modifying or replacing workflow definitions. However, this is not possible when languages in use cannot support the new requirements, or can only support them in a complicated or counterintuitive way. Various examples of these situations can be found in the literature on workflow patterns, which shows how languages can or cannot support common workflow structures [vdAtHKB03, RtHvdAM06, RvtE05, RtHEvdA04, RtHEvdA05, RvdAtH06]. For instance, if a BPEL-based application needs to model a *structured discriminator*[5], it has to do so in an extremely complicated way because BPEL does not support the pattern [WvdADH03]. In cases where the language cannot support a certain requirement, the solution is to adapt or extend the language. However, this implies modifications to the tools that are part of the WfMS and depend on the language, such as the editors, the engines, and others.

Requirements that change are not the only reason to modify a language. Applying an existing language to a new domain may also require modifications to it. For instance, BPEL includes many features to model and enact business processes. However, it lacks the concepts related to management of people, such as roles and task assignments. Therefore, an extension to the language was created, which introduced all those concepts and related them to existing BPEL elements (BPEL4PEOPLE [AAD$^+$07]).

Unfortunately, there are also cases where existing languages are not compatible at all with the new requirements or with the new domains. In those cases, it is not enough, or it is not cost effective, to modify an existing language. For instance, modifying BPEL to include *arbitrary loops* [vdAtHKB03, RtHvdAM06] would require a total redefinition of the language. In those situations, new languages are designed and implemented. Of those, many are Domain Specific Workflow Languages.

Creating and supporting a new language has both advantages and disadvantages. In the first place, there is the high cost associated to the design of the language and the development of the corresponding tool chain. In particular, supporting a new language usually requires, at least, the implementation of a new editor and of a new workflow engine. The construction of a workflow engine is a critically expensive step to support a new language [Nut96]. Another disad-

[5]In the *structured discriminator pattern*, there is a *join* of two or more branches, that follows a corresponding *split* earlier in the process. When the first incoming branch is enabled, the control flow is passed to the part of the process that follows the join. Subsequent enablements of the incoming branches, do not result in passing the control flow. The discriminator is reset when all incoming branches have been enabled [Wora].

vantage of creating a new language is having to train users to use it. Depending on the characteristics of the language, this can be a difficult task. In particular, DSWfLs should offer a better user experience and should be easier to learn for domain experts [WEB+07, Dee07]. Finally, another aspect to consider is that of maintenance, since introducing a new language also implies having to maintain a new tool chain. On top of that, the more specific languages are, the more likely they are to change [Cle07a]. Thus DSWfLs need to be designed from the beginning with growth and change in mind.

Developing new workflow languages also brings advantages, especially when they target particular domains. In the first place, when a new language is designed, the elements included in it are the ones most suitable for the problems that have to be solved with the language. In many cases, these elements are aligned to domain concepts and thus they capture better domain knowledge. Furthermore, these languages are usually smaller and simpler than GWfL, which makes them simpler to implement, maintain, and modify [vDK98].

1.2 Problem statement

The problem addressed in this dissertation has two aspects. The first one is the need *to support extensions and adaptations to existing workflow languages.* Currently, this is impeded by the characteristics of workflow engines and languages, which have not been designed with flexibility as one of their most important capabilities. The second aspect is the need *to support new workflow languages, and, in particular, domain specific workflow languages.* This is also problematic because it requires the development of a number of tools, including editors, clients, monitoring applications, and, most importantly, workflow engines [Nut96]. This dissertation addresses a solution for these problems, but limited to the problems affecting workflow engines. Solving these two problems for the rest of the tools in a WfMS is outside the scope of this dissertation, and even though its results can act as a solid basis, further research is required to address these issues.

1.2.1 Challenges to the development, extension, and adaptation of workflow engines

Currently, there are a number of conditions that challenge the creation of tools to support workflow languages, and also limit the extent to which they can be adapted and extended. In this section we discuss these conditions.

Limited extensibility and adaptability of engines and languages

Many workflow engines currently in use were not designed considering flexibility of the languages[6] as a fundamental requirement. On the contrary, they were designed and optimized to run a single version of a specific language. Some of them can handle a few extensions, but these are usually very limited. This situation is evidenced in the architecture of those engines, which is not suited to accommodate changes to the workflow specification languages. Therefore, it is very difficult to introduce modifications and support new, unexpected requirements.

[6]"Flexibility is the ease with which a system or component can be modified for use in applications or environments, other than those for which it was specifically designed" [IEE90].

In the cases where workflow languages offer flexibility points, these are fairly limited. They usually allow just the specialization of some of the elements. Only in a few cases it is also possible to add new elements, albeit with restrictions. However, the adaptation of existing elements is never possible. As an example, the only flexibility points in BPEL are the tag `<extensionActivity>`, and the possibility of supporting some extra attributes and elements. Similarly, BPMN offers some extensible elements called *artifacts*, while the specification of XPDL mentions *vendor or user specific extensions.*

An additional problem is that the flexibility offered by these languages does not necessary reflect on their engines. Therefore, it is not always feasible to extend the languages because the engines may not support the extensions. These drawbacks can be seen in BPEL4PEOPLE (B4P) [AAD$^+$07], one of the best known BPEL extensions, which offers a very expressive way to introduce human interactions into BPEL processes. B4P makes use of the extension points of BPEL to introduce complex people-related information and new types of activities. However, since the extension of existing BPEL engines to support B4P is not trivial [HVD08], B4P is not currently supported by many BPEL engines.

In summary, current workflow engines offer very limited capabilities for extension and adaptation, or offer no capabilities at all. Because of this, changes to workflow specification languages require either complex and extensive modifications to engines' implementations, or require complete re-implementations.

Misalignment between high level model information and low level run time information

Another factor that challenges the flexibility of workflow engines and languages is the misalignment between *i*) the structure of the languages, and *ii*) the structure of the engines' implementation and the associated run time information. For instance, a workflow definition in OPERA[7] is expressed with a high level language, but it is usually very different from the run time representation that is expressed with OCR or with database specific schemas. As a result, it becomes difficult to map changes to the languages directly onto changes to the implementation. Furthermore, as changes are introduced, the relation between implementation artifacts and language elements may become more difficult to understand and maintain [Eva03].

In addition to the impact on engines and languages' flexibility, this misalignment also affects the other tools that depend on the language and interact with the engine (e.g. clients and monitoring applications). The main problem with those complementary applications originates in their high coupling with the engines, and the dependency they have on the languages. In this case, the complex mapping previously discussed appears again: changes to the languages reflect on difficult-to-trace changes to the engines, and those reflect on changes to the tools. In the end, the relation between specific changes to the languages and specific changes to the tools is complex to follow.

On the contrary, if languages and engines' implementations were well aligned, it would be very easy to introduce and track changes in the languages, the engines, and the tools.

[7]Section 2.4.2 presents more details about OPERA, a kernel for distributed workflow execution based on a canonical representation model (OCR).

Limited support for developing new engines

Another challenge in solving the problem of building new workflow engines is the relative lack of frameworks or libraries to support their development. Because of this, implementing a new engine currently represents a substantial amount of work.

In the past, a few frameworks and kernels have been designed to support the development of workflow engines. However, these frameworks usually present two important restrictions that limit their applicability:

1. *Prescribed execution models*

   Several frameworks and kernels have an embedded execution model that cannot be modified. Because of this, these tools can only be used with languages that are compatible with the provided execution model. For instance, the Process Virtual Machine – PVM [BV07] supports only one execution model based on hierarchical tokens. Consequently, to implement an engine on top of the PVM, the semantics of the language have to be described in terms of that token-based system.

   In order to avoid this problem, kernels and frameworks should offer mechanisms to adapt or replace the execution models. Some of the possible adaptations include changing the behavior of existing concepts, or introducing new ones. Unfortunately, this is only offered by a few kernels. Section 2.2.2 discusses the role of the execution models in workflow systems, and section 2.4 presents various workflow kernels, including a few that support changes to the execution models.

2. *Fixed intermediate workflow languages*

   Several workflow kernels and frameworks are based on the usage of an intermediate workflow language. The idea behind this strategy is to implement the necessary elements to support just one executable language, and then transform various high level languages into this intermediate one. Usually, this intermediate language is low level in order to be more generic. This strategy is used, for example, to execute BPMN processes in BPEL engines. In this case, BPEL can be considered the intermediate workflow language.

   There are various kernels based on using intermediate workflow languages. These include the 'Workflow Kernel' of Ferreira and Pinto Ferreira [FF04], the work of Fernandes et al. [FCS04], OPERA [AHST97b, AHST97a], and Mentor [WWWKD96]. Section 2.4 presents these, and a few others, in more detail.

   However, there are three issues associated to the usage of a fixed intermediate workflow language. The main one is that the expressive power of the workflow language is limited by the expressive power of the intermediate language. Therefore, it is possible to have structures in the newly envisioned language that cannot be represented in the executable one. In those cases, either the original language has to be constrained, or the intermediate language has to be changed. In BPMN, the first alternative was selected: the mapping from BPMN to BPEL described in the BPMN specification only includes a subset of BPMN's elements.

> The second issue is the additional complexity introduced by the mapping from one language to the other. This mapping is usually implemented with transformations which leave traceability information that has to be interpreted to relate the elements from those two languages. For instance, this information is used to relate run time events and state changes in the executable models, with events and state changes in the original, high level model.
>
> Finally, there is an issue that arises when languages are modified. In such cases it is also necessary to change the transformations. However, finding the transformations affected by the changes, building the mapping, and maintaining it can be complex and expensive [KKS07].

Engines are tightly coupled to languages

A related challenge is the tight coupling between workflow engines and workflow languages. This means that engines are designed to support a specific language, and that all the elements of its implementation are aligned with that particular language. This coupling makes it very difficult to use an existing engine to execute a new workflow language, and it is related to the limited capabilities of extension and adaptation.

As an example of the disadvantages posed by this challenge, one can consider the case of BPEL. Most BPEL engines are very complex applications which are integrated in workflow management systems and interact with several different applications. The complexity of these engines is not limited to the complexity inherent to executing BPEL processes. Instead, this complexity arises from additional features such as the communication infrastructure, the management of persistence, the scalability of the system, and its reliability. The applications that interact with a BPEL engine include debuggers, monitors, and clients of different kinds. Because of all the features offered by BPEL engines, BPEL would seem a very reasonable choice for use in any domain. However, experts in different domains have found limitations of BPEL and then have opted for developing their own languages from scratch (e.g. Taverna [HWS$^+$06, OGA$^+$06]). These experts have had to develop their own engines because of the tight coupling between BPEL engines and the language. As a consequence, they have re-implemented many of the features offered by BPEL engines, and adapted those features to their own languages.

A related example is Sedna [WEB$^+$07], which extends BPEL. Although the language is not compatible with BPEL, its developers built the tools to enable its execution using BPEL engines. To achieve this, they built a new editor capable of transforming Sedna's definitions into BPEL, and a monitoring tool that converts run time BPEL information back into Sedna.

Tool chains are not portable between languages

Continuing with the previous example, developers of new languages also face the need to develop new tools to interact with their new engines. Because they have not been able to reuse existing engines, in most cases they have also lost the possibility of using existing tools. Therefore, they have had to develop their own tool chains.

We consider the need to re-develop entire tool chains an important challenge to the development of new languages. If existing tool chains could be more easily reused with new languages, then new and more adequate languages would be developed.

1.2.2 Objective

The objective of this dissertation is to solve the problems identified with a novel approach to support the development of engines for new workflow languages, as well as to support their extension and adaptation. The proposed approach includes the necessary abstractions to define the languages, and a framework to implement the corresponding workflow engines based on a novel architecture. To guide the usage of these elements, we also present the main elements of a process to develop, adapt, or extend workflow engines.

The following are the four characteristics that we consider central to overcome the mentioned limitations and achieve the stated objective. These characteristics are present in our proposal, and they were a central consideration to the design of the whole approach and the platform that implements it. We believe that any other solution that presents the same characteristics is also likely to solve the stated problems.

C1. The platform is independent from particular workflow languages and execution models

To support the largest possible number of languages, the solution that we present in this dissertation is not based on elements from particular workflow languages or workflow execution models. Instead, the structure and semantics of language elements are expressed using more generic elements because every restriction introduced can potentially limit the number of languages supported. As we show later, to avoid these restrictions we embraced, to a certain point, the usage of general purpose programming languages. On the other hand, this also means that fixed intermediate languages (like those used by Ferreira and Pinto Ferreira in [FF04], or OCR in OPERA [AHST97b]) are not an acceptable solution because they could limit the ability to support certain languages.

The elements that support the definition of languages have specific characteristics to support features normally required in workflow languages and engines. As a result, the proposed platform supports requirements such as the following: allowing the concurrent execution of multiple tasks and controlling their synchronization; allowing the parallel execution of multiple instances of the same process that involve different participants and data; handling the interaction between external systems and the running workflow instances; allowing the execution of tasks that interact with external applications; allowing the persistence of the processes state.

The variability found in current workflow languages makes it impossible for a single platform to claim that it can support *any* workflow specification language, and it only gets worse when DSWfLs are considered. Instead, we analyzed several existing languages and workflow execution models to identify their most relevant characteristics, and we aimed to support those characteristics in our platform. This created a reasonable scope which, as the case studies presented in chapter 7 evidence, includes both generic and domain specific workflow languages, and includes several commonly used workflow execution models. The trade-off of this is excluding some workflow languages that require uncommon features. For example, our solution does not target distributed workflows [PPL01], rule based workflows [WLC$^+$05, GKD01], and logic based workflow languages [GP07, DM05].

C2. The platform maintains a clear mapping between language elements and implementation elements

The platform presented in this dissertation provides a way to easily establish a mapping between elements of the language and elements of the implementation. Furthermore, in the event of language evolution, this mapping is easy to maintain. The main benefit taken from this characteristic is facilitating the evolution and maintenance of the languages, by avoiding the misalignment between languages and their implementation.

C3. The platform supports language flexibility

The proposed approach favors the extension and adaptation of workflow specification languages. This is achieved because the platform offers a series of extension mechanisms that are available to all the workflow languages built on top of it. Therefore, extensions and adaptations do not have to be done in an ad hoc way, they can instead be based on common extension operations.

As in the case of characteristic C1, our platform does not try to support every imaginable extension for each supported language. Instead, we have built a taxonomy of composable extensions and adaptations that the platform can support (see chapter 4). These basic modifications can be composed to achieve more complex results, such as those illustrated in chapters 4 and 7.

C4. The platform is reusable and supports the implementation of open engines

Workflow engines are not isolated applications, but are instead frequently found embedded in complex systems known as workflow management systems. Therefore, workflow engines need some means to interact with other elements of the same system, to interact with external elements, and to react to requests coming both from inside and outside the system. To achieve these requirements, our platform supports the development of *open engines*. This means that the internals of each engine are observable by external components and systems. Additionally, some of its elements can also be controlled to some degree from the outside.

On top of that, the platform is reusable and adaptable. Therefore, a basic set of elements are shared by every engine, while some other elements are adapted to the needs of each particular language. Together, these two characteristics contribute to solving the problem of porting or re-developing the tool chains associated to the languages. Since the engines are open and offer powerful interfaces, complex tools can be developed to interact with them. On the other hand, since the platform is reusable, all the tools can be programmed using a common API. This opens the possibility of using some of those applications with several languages. This possibility cannot be guaranteed because it depends on the characteristics of each application, but currently it is not available for most workflow engines and tool chains.

1.3 Thesis approach

To achieve the stated objective, in this dissertation we propose *a novel architecture to build workflow engines, based on metamodels and on the modularization of workflow definitions.* By using this architecture, the development of engines for new workflow languages does not have to start from scratch. Instead, engines are built on top of a *reusable workflow kernel* that can be configured to support different workflow languages. Furthermore, the engines built using this kernel are *more adaptable and more extensible* than currently available engines.

The main elements and ideas of the proposed approach are the following:

- A metamodeling platform for modeling workflow definitions.
- Modularized workflow definitions using concern specific workflow languages.
- Run time coordination of executable workflow models.
- A platform based on Open Objects.

In the following we will briefly explain these key ingredients of the approach that we have implemented in a platform called *Cumbia.* In later chapters we will provide a more detailed discussion on each aspect of the approach.

A metamodeling platform for modeling workflow definitions

The reason behind some of the problems and challenges discussed is the difficulty to localize the elements in the engines' implementations affected by changes to the syntax or the semantics of the languages. These aspects are often scattered around in the engines' code, and they are entangled with the implementation of other functionalities, thus making languages very difficult to modify or replace. To counteract this, our approach offers the means to separate the implementation of the languages from the implementation of complementing functionalities. By doing so, the people in charge of implementing the semantics of the languages, do not have to implement those other functionalities, the implementation artifacts become more coherent, and the languages are more easily replaceable and modifiable.

Using metamodels to represent languages is a technique frequently applied in several contexts [GKP98]. Often, only the abstract syntax of the languages is modeled, but some metamodeling platforms can also model the semantics of languages, or parts of it. The level of abstraction and the level of detail included in those representations depends on the characteristics of the platform. Figure 1.2 shows how metamodeling techniques can be applied to the workflow context. A *metamodel definition* is constructed to represent a *workflow specification language.* Then, *model definitions* conformant to the metamodel are built to represent *workflow definitions.* Finally, *workflow instances* (also known as *cases*) are represented with *model instances.*

Figure 1.3 provides a graphical representation of the main elements in our proposal to support the definition and enactment of workflow specification languages. Since a metamodeling approach is used, the first element to consider is the *metamodeling framework.* It defines the abstractions to build *metamodel definitions* and represent the structures and semantics of the languages. This

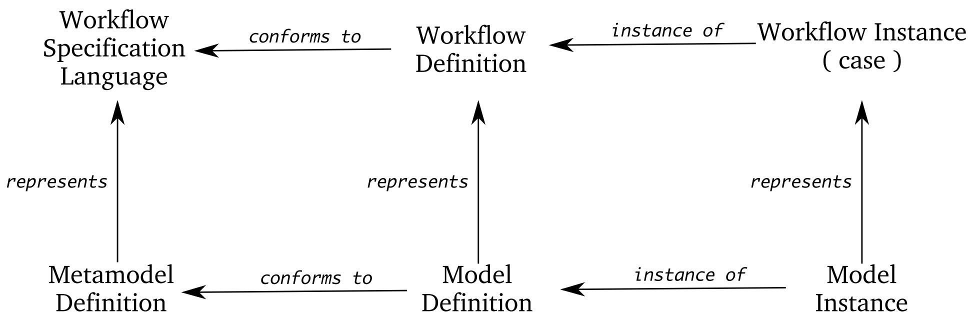


Figure 1.2 Metamodeling and workflows

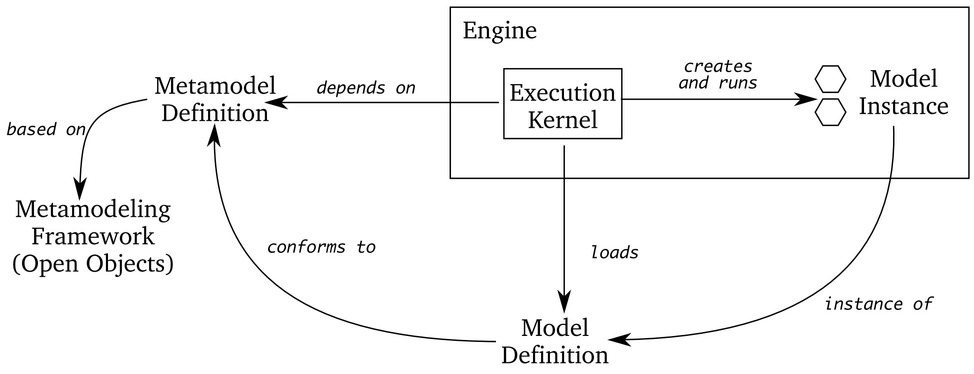


Figure 1.3 An execution platform based on metamodels

dissertation proposes *open objects* as the main modeling abstraction to use in the metamodeling framework.

The second element in the proposal is a component called *Execution Kernel* (EK). The most important of its multiple responsibilities is to enact *model definitions*. To support this, the EK offers the means to load model definitions, instantiate them, and coordinate the run time behavior and interaction of instance elements. Nevertheless, the EK is generic and it requires access to the information contained in the metamodel definition to be aware of the elements, the structure, and the semantics of the enacted language. Additionally, the EK holds other responsibilities, such as managing persistence, doing basic conformance verifications, offering interfaces to allow external interactions with the running model instances, and generating events to facilitate monitoring. By using the EK as the foundation of every engine, the features it offers do not have to be redeveloped for each new language. As a result, the effort required to support a new workflow language is considerably reduced, thus making the development of domain specific workflow languages more likely.

Compared to existing workflow kernels, the one we are proposing is more flexible and more expressive because there is not a fixed underlying workflow language or workflow execution model. Instead, all the aspects of the new languages can be defined from scratch, restricted only by the characteristics of the abstractions provided by the metamodeling framework. Later chapters in this dissertation demonstrate that these restrictions are less problematic than the restrictions found in other kernels based on intermediate languages and models. In those, the semantics of every workflow language has to be reduced to the semantics of a unique workflow execution model which is not always well aligned with the necessities of the language.

This aspect of the approach also has a positive impact on the maintainability and extensibility of the engines. By using metamodels to structure the languages and the implementation of their semantics, the decomposition of language constructs is preserved in their implementation. This makes it easy to maintain a mapping between language elements and implementation elements. Furthermore, it makes the implementations easier to understand, maintain, extend, and evolve [Cle07b].

Finally, there is another advantage associated to using one workflow kernel to implement many workflow engines. Typically, workflow engines are developed in an *ad hoc* fashion and there are no commonalities between them at the code level. Therefore, the associated tool chains are also developed in an *ad hoc* way, and thus are difficult to reuse. On the contrary, if the same kernel is shared among various workflow engines, the reuse of complementary tools is more likely to occur. Towards the end of this section we discuss how this is complemented with the usage of open objects to achieve the goal of having a reusable platform for open engines.

Modularized workflow definitions using concern specific workflow languages

Currently, most workflow languages, and especially generic workflow languages, are monolithic and tend to include a large number of elements, which are not always closely related. These languages offer only simple modularization strategies that are generally restricted to the hierarchical decomposition of processes. These factors have had negative impacts on the languages. On the one hand, bigger languages tend to be more difficult to understand and learn. They are also more difficult to maintain, and because of the lack of powerful mechanisms of modularization, the definitions tend to be large and complex. Therefore, they are more difficult to create, maintain, and reuse. In some domains, these limitations are very inconvenient.

On the other hand, the aforementioned factors have also limited the extensibility and adaptability of the languages. Since languages are defined in a monolithic way, it is not easy to replace parts of it. Similarly, it is not easy to add new elements and extend the original languages.

In recent years, some modularization strategies for workflows have been proposed. In particular, Padus [BVJ$^+$06] and AO4BPEL [CM06] have applied aspect-oriented techniques to workflow modularization. Nevertheless, these approaches have only solved part of the problems by offering strategies for the modularization of workflow definitions. As a result, the issues with the workflow specification languages, such as their size and complexity, have not been solved.

The approach that we propose employes a modularization strategy based on concern separation and inspired on multimodeling techniques [BCF$^+$08]. This strategy has two sides to consider. The first one, is that workflow definitions can be decomposed according to concerns, which is similar to what Padus and AO4BPEL propose. The result is depicted in figure 1.4, where a monolithic workflow definition is split into a set of concern-specific definitions. These definitions are complemented by composition information, which serves to relate the otherwise totally separated parts. We call this set of definitions, together with the information that relates them, a *workflow assembly*. The second side of the modularization strategy is that concern specific languages can be used to specify each part of an assembly. Therefore, these languages can be very expressive and employ high level elements adapted to each concern.

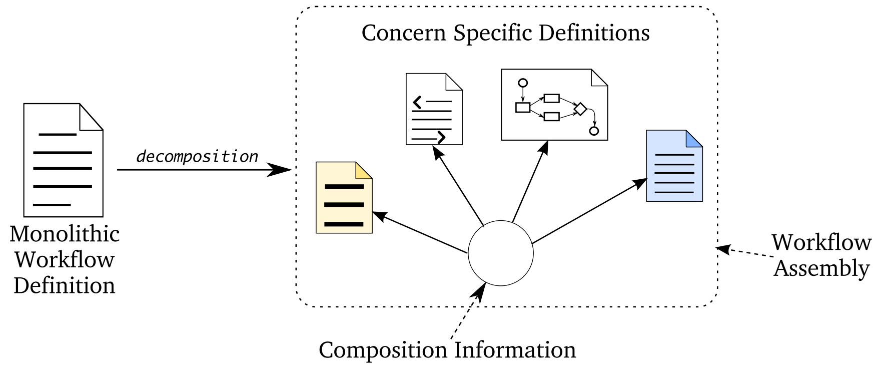


Figure 1.4 Concern specific workflow languages and definitions

There is not a single technique to identify the concerns to include in an application. For example, they can be identified based on functional elements of the process, such as business, billing, or auditing activities. Another example is to follow the traditional decomposition in perspectives (i.e. control, resources, data, etc. [vdAvH02]). The only requirement is that each detail that is relevant for the processes must be included in a concern.

Run time coordination of executable workflow models

When concern specific workflow languages are used to describe a workflow, one or more models are built for each involved concern. Afterwards, these models have to be assembled to reconstruct the semantics of the complete workflow. In our approach this is achieved with a model weaving technique that depends on links described between *model definitions*. This means that the composition problem is solved at the model level and not at the metamodel level. Metamodels are thus completely independent and there is nothing to explicitly relate them. By doing so, the coupling between languages is lowered, and this favors their evolution. The downside of this is that the relations between models have to be written for each assembly.

The other characteristic of our approach regarding the composition of workflow definitions is that they are never combined. Instead, lightweight links are established between model instances, and these are used to coordinate their execution. This approach is very different from most model weaving techniques, which combine the source models to generate a composed one. On the contrary, our approach keeps concern specific models related in such a way that they can interact, while at the same time they are recognizable and have well defined borders and structures. The weaving between model instances is established after creating them, but before starting their execution, and it serves to coordinate their execution.

Additionally, the links established between model instances are modifiable at run time. Therefore, assemblies are not totally fixed, and their structure can change even after they have been instantiated. This additional degree of flexibility complements the possibility of changing the structure of model instances at run time. These two features are very important for some domains, such as e-learning.

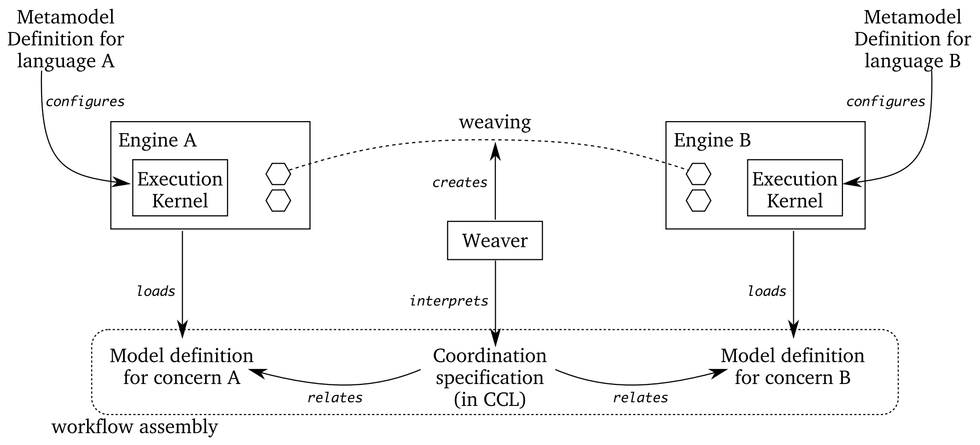

Metamodel Definition for language A
Metamodel Definition for language B
configures
configures
weaving
Engine A
Execution Kernel
Engine B
Execution Kernel
creates
Weaver
loads
interprets
loads
Model definition for concern A
Coordination specification (in CCL)
Model definition for concern B
relates
relates
workflow assembly

Figure 1.5 Coordination of workflow concerns

In order to coordinate a set of models, information additional to the models' structure has to be provided. In Cumbia, this information is a weaving program written in a language called CCL (see figure 1.5). This language, the *Cumbia Composition Language*, is independent of the concern specific languages. Therefore, a new composition language is not required when new languages are introduced.

A platform based on Open Objects

In order to implement a platform with the characteristics previously mentioned, we developed a notion called *open objects*. Open objects are the main modeling abstractions to build and implement the metamodels. When a metamodel for a workflow language is defined in Cumbia, each element in the language has to be modeled with an open object. In this way, the open object encapsulates the structure of the element and the behavior it should have when used in a model. Open objects also externalize their interaction with other elements in order to make this interaction flexible.

There are two main characteristics of open objects that make them useful to implement the proposed platform. In the first place, open objects offer various coordination mechanisms, both synchronous and asynchronous, that can be used to control the interaction between elements in the models. These mechanisms make the execution of the models possible, and also allow the coordination of various concerns. CCL and the weaver also rely on those mechanisms, as CCL instructions are expressed in terms of open objects' features. The interaction mechanisms of the open objects, together with the externalized state machines and powerful interfaces to control and observe them, makes this notion central for the support of open engines.

On the other hand, open objects offer various mechanisms to extend and adapt metamodels. Each one of those mechanisms represents a different kind of change that can be applied to an open object. For example, some of these mechanisms serve to specialize the behavior of an open object, or to modify the way it interacts with other elements. By stacking up multiple of those changes, complex adaptations and extensions to the languages can be supported.

1.4 Thesis contributions

The following are the main contributions of the research presented in this dissertation.

- **A novel architecture for workflow management systems**

  This dissertation proposes an approach to build extensible and adaptable workflow engines, using an architecture that favors their evolution and maintenance, and favors the evolution of the workflow languages that they support. On the other hand, this architecture also targets the creation of open engines, thus facilitating the integration of complementary tools, such as clients or monitoring applications.

- **A novel approach for workflow modularization**

  This dissertation explores a multimodeling approach to modularize workflow definitions and workflow specification languages. This approach is based on the identification of relevant workflow concerns in each domain, the design and implementation of concern specific workflow languages (CSWfLs), and the modularization of the workflow definitions accordingly to the concerns identified. This approach should result in CSWfLs that are simpler to use and are more suitable than monolithic, generic workflow languages. On top of that, the modularized definitions described with those languages should be easier to write, understand and maintain.

Additionally, this dissertation provides the following secondary contributions.

- **Open Objects, a base element for executable modeling**

  This dissertation defines the notion of open object, and uses it as a base element to build executable models. The usage of open objects results in metamodels and models with coordination features and flexibility characteristics pertinent in a workflow context.

- **The Cumbia Platform**

  This dissertation presents the Cumbia platform, which implements the proposed approach. The central element in the platform, the Cumbia Kernel, supports the execution of open objects, and can be used as the base to develop engines for new workflow languages. Other important elements of the platform include CCL, a language to describe the coordination between concerns of a workflow definition, and the Cumbia Weaver, which interprets the CCL programs before the execution of the workflows.

- **Validation of the Cumbia Platform in workflow applications based on MiniBPMN, YAWL, Petri nets, IMS-LD, BPMN, BPEL, and other workflow languages.**

  The proposed approach and the Cumbia Platform is validated with the implementation of several languages with varying characteristics that test different aspects of the approach.

1.5 Organization of the dissertation

The following is the structure of the rest of this document.

Chapter 2: Workflow Modeling and Enactment, presents an overview of the workflow context and the main characteristics and limitations of current modeling techniques and execution tools. The first part of the chapter is devoted to modeling, and thus it discusses languages, execution models, and modularization strategies. The second part of the chapter focuses on workflow execution, and presents some characteristics and limitations of representative tools that are used for workflow enactment, or serve to develop workflow engines.

Chapter 3: Executable Models in a Workflow Kernel, presents the main characteristics of the solution and shows how these characteristics contribute to solve the problems identified in chapter 1. The chapter is concluded with an overview of the implementation of the solution in the Cumbia platform.

Chapter 4: Workflow Models based on Open Objects, presents the details about the Cumbia platform and explains how it can be used to support a workflow language. This chapter describes core concepts of our approach, such as the open objects and their usage for metamodel definition. The chapter uses a language called `MiniBPMN` to illustrate the ideas discussed: this language is concern specific and only includes elements to describe the control flow of a process.

Chapter 5: Coordination of Multiple Concern Specific Models, complements chapter 4 by showing how several concern specific languages are supported in Cumbia. To illustrate this, the dimension of time is identified in a workflow application, and a metamodel is designed to describe it. Then, the Cumbia Composition Language (CCL) is used to establish relationships between control flow and the time restrictions, thus achieving the initially intended execution semantics.

Chapter 6: Towards a development process for workflow engines based on Cumbia, presents guidelines to organize the development. On the one hand, it identifies activities to gather information about the problem at hand and develop the necessary artifacts using what Cumbia offers. These activities go from selecting or developing workflow specification languages, to executing models described with said languages. On the other hand, this chapter characterizes the stakeholders of the development process using their skills and knowledge. This chapter also presents some tools that are additional to the core of the proposal and facilitate the usage of the core of Cumbia.

Chapter 7: Validation, presents the application of the Cumbia platform in various case studies. These scenarios are different from the examples presented in chapters 4 and 5 and they aim to cover an interesting part of the spectrum of workflow languages. The languages selected for these case studies include pre-existing ones, and also ad hoc languages that we developed. The scenarios presented in this chapter also differ because they target different workflow domains.

Chapter 8: Conclusion, concludes the dissertation with the presentation of the most important conclusions, a revision of the contributions, and a brief discussion of future work.

2 Workflow Modeling and Enactment

The domain of workflow-based systems has seen the development of different technologies and strategies to implement and adopt workflows. Among these things, there are several execution models and hundreds of workflow specification languages based on them. Consequently, workflow engines to execute these languages have been developed. However, many of these engines have been constructed mostly from scratch, in an ad hoc fashion. One of the reasons behind this is the relative lack of frameworks to develop engines for new languages. There are a few frameworks available to alleviate this situation, but some have limitations, while others target too specific domains.

This chapter presents a brief overview of some of the developments that were just mentioned. In order to do so, the first section introduces the basic vocabulary and concepts about workflows, which are used in the rest of this chapter and the dissertation. The following section focuses on workflow modeling and illustrates various workflow specification languages and workflow execution models. The final part of the section presents two metrics to compare and select workflow languages and execution models: expressive power and suitability. The following section introduces the topic of workflow modularization and presents the most relevant works in this area. Afterwards, section 2.4 discusses various works that have addressed the problem of supporting new workflow languages. The approaches that we present in this section offer a comprehensive view of the available solutions to tackle the issue. Finally, section 2.5 revisits the challenges presented in the introduction and shows that the options currently available to develop engines for new languages, have several limitations.

2.1 General workflow concepts

Workflows are a technology that has been in development for over thirty years. Workflows had their origin in the development of computational systems to handle office procedures in paperless offices, but soon their usage spread to many other domains [GHS95, EN96, zM04]. Nowadays, workflows are used in contexts as dissimilar as business process modeling (BPM), e-learning, scientific applications, computer aided engineering, and health services management.

In general workflow systems are used for one of the following objectives [CKO92]:

- To facilitate *understanding* and *communication* of workflows in an organization.
- To support *process management*, by offering the possibility of monitoring, controlling and adjusting the workflows in real time.
- To support *process improvement*, by offering tools to gather performance information and adjust the processes based on its analysis.
- To provide *automated guidance.* This means providing suggestions to users that participate in a workflow, to facilitate their work and to improve compliance to the workflow definition.
- To *automatize workflow execution.* This objective has two parts. The first one is executing the software-based steps of the workflow. The second part is supporting the interaction of humans at the right points of the execution.

From these objectives it emerges that the essential feature of workflow technology is its focus on describing a series of actions, and describing the order to perform them. Depending on the specific workflow technology, the actions can be of various kinds. For example, in a language like WS-BPEL[1] actions usually serve to consume web-services or to transform xml data [OAS05]. On the other hand, in IMS-LD[2] actions usually involve the participation of users that have to read provided material, answer tests, and other learning activities [IMS03b]. The order in which actions are performed can also be specified in different ways. For example, it is very common to specify an explicit flow of control, although some tools can also specify dependencies between actions. In the former case, the specification says what has to be done *after* each action is completed. Languages such as BPMN[3] use *flows* to describe this. In the latter case, the specification says what must happen before an action can be executed. For instance, these dependencies can be described with temporal logic [DM05].

Another common characteristic of most workflow systems is using specialized languages to describe *workflow definitions.* These languages are commonly known as workflow specification languages, workflow definition languages, or simply workflow languages. In the literature, the term ‘workflow’ has been used both for the *real life process* and for its representation in a workflow system. To avoid the confusion, we will use the term ‘workflow definition’ for the representation of a ‘workflow’.

Many workflow specification languages have been designed to be used by non-technical users, and they are usually expected to be high level. However, many workflow languages use low level concepts (e.g. WS-BPEL’s XSLT transformations of XML data), and there are even workflow languages that are similar to general purpose programming languages (e.g. Workflow Prolog [GP07]). Section 2.2.1 presents the basic details about some common workflow languages which illustrate part of the available spectrum.

[1]WS-BPEL, or BPEL, will be described in section 2.2.1
[2]IMS-LD will be described in section 2.2.1
[3]BPMN will be described in section 2.2.1

A workflow definition can be used for several purposes. For instance, it can be used just as documentation of a fully manual process. Another example is using the specification as an input for analysis tools which try to discover inconsistencies in the specifications. Nevertheless, for this dissertation the most relevant usage for the specifications is as the input for workflow engine which *enacts* them. According to the definition of Grefen and de Vries, enacting or executing a workflow means processing workflow events and initiating reactions to these events as defined in the workflow definition [GdV98]. For example, in the context of BPMN, the finalization of the execution of an activity is a workflow event; the reaction to this event is to start the execution of the ensuing activities.

Workflow definitions can only be enacted if the semantics of the languages are well defined semantics and implemented in the engine. These semantics are usually defined in terms of a known execution model, such as those described in section 2.2.2. In each case, these execution models are tailored to include behavior specific to the language. For example, the way to invoke concrete activities or the supported data transformations. Nevertheless, the most complex behavior, which usually involves aspects such as managing concurrency and synchronization, is usually defined in the execution model itself.

Another important concept related to workflow execution is that of *workflow instantiation*. A workflow instance, or *case*, is the representation inside a workflow system of a single enactment of a process [Coa99]. Usually, each workflow instance is capable of independent control, and maintains its own internal run time state and data. In some cases, workflow instances can also share data at the specification or at the engine level.

Since many workflows support interactions with external elements, it is important for the engine to have mechanisms to differentiate the instances of a workflow definition. This is usually achieved with a unique identifier, but in some cases this can have a big impact and require more complex mechanisms. For instance, the necessity for having case identifiers in YAWL was one of the factors for using Extended Workflow Nets instead of Petri nets [vdAtH06]. Another example is BPEL, where *correlation sets* uniquely identify process instances, and are required to support asynchronous messages.

In addition to the execution itself, which includes the invocation of external applications and the management of the resources consumed and produced during said execution, workflow engines in different contexts support some different sets of additional requirements. One of them is *monitoring*, which refers to the activities performed by users or external systems to trace the run time state of the enacted workflows and the produced data. Monitoring is usually done through applications connected to the engines, which query the state of the workflow instances, or capture relevant events.

Another example is what some workflow engines call *dynamic adaptation*. This term refers to the capacity to modify workflow instances or specifications, at run time. In general, this is complex to support, especially because it is necessary to adapt the run time state of the instances to the new structure. Depending on the engine and the particular requirements of each language, different strategies are followed to implement this. As an example, some engines only support some predefined kinds of changes. On the contrary, others are much more flexible but offer less guarantees about maintaining the consistency of the instances. This also means that not every change is supported in every workflow engine or has the same impact. For example, some engines only support changes to selected workflow instances, while in others changes can be applied to all the instances of a workflow definition.

Workflow engines are seldom isolated from other applications. On the contrary, they are usually part of a package known as a Workflow Management System (WfMS) which also includes tools to define the specifications, to control and monitor their execution, and to allow the interaction of participants and applications [Coa99]. The *workflow reference model* of the Workflow Management Coalition proposed a set of interfaces to standardize the elements of a WfMS and facilitate the interaction of tools from different vendors [Hol95, Hol04]. Figure 2.1 shows the main components and interfaces proposed by this reference model.

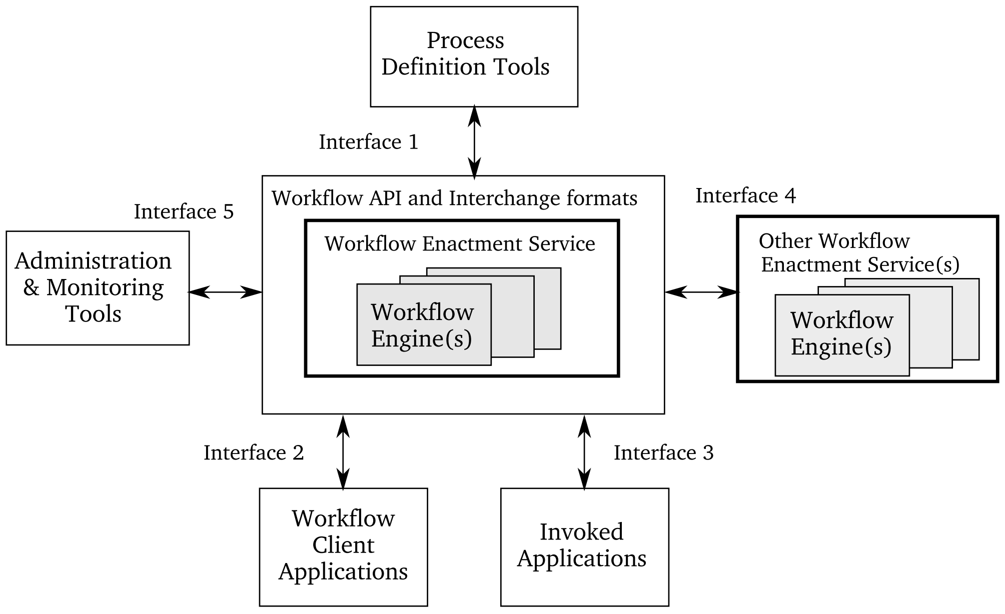

Process Definition Tools
Interface 1
Workflow API and Interchange formats
Interface 5
Interface 4
Administration & Monitoring Tools
Workflow Enactment Service
Workflow Engine(s)
Other Workflow Enactment Service(s)
Workflow Engine(s)
Interface 2
Interface 3
Workflow Client Applications
Invoked Applications

Figure 2.1 Components and interfaces defined in the Workflow Reference Model [Hol95]

In the center of the figure there are workflow engines which expose a common workflow API based on standardized interchange formats. The API defines the following five interfaces that serve to interact and exchange information with different kinds of applications:

- **Interface 1: Process definition tools.** This interface is used by process definition tools (editors) to deploy workflow definitions into engines. The format proposed by the Workflow Management Coalition to use for this is XPDL.

- **Interface 2: Workflow Client Applications.** This interface is used by applications that mediate in the interaction between users and running workflows. For instance, one such client application can request information from a user and then give that information back to an activity in order to process it and continue with the workflow execution.

- **Interface 3: Invoked Applications.** This interface in the reference model is used to allow back and forth interaction with external applications that participate in a workflow.

- **Interface 4: Other Workflow Enactment Services.** In the workflow reference model, interface 4 is described as the interface to allow the interaction between several instances of a workflow engine, or between several different workflow engines. This was an important requirement for the Workflow Management Coalition because of their interests in achieving compatibility between engines and achieving the integration of existing systems.
- **Interface 5: Administration and Monitoring Tools.** The fifth and final interface described in the reference model allows the interaction of administration and monitoring tools with the engine. These tools have two main objectives. The first one is to control and manage the behavior of the engines and the execution of the workflows. The second one is to gather and display information about the running workflows.

In spite of its advantages, the workflow reference model was never adopted in full. This happened mainly because it imposed tight requirements that implied profound changes to existing tools. For instance, the adoption of XPDL as the sole intermediate workflow language was seen as an impediment to the development of workflow specification languages.

2.2 Workflow modeling

In order to model a workflow, it is necessary to first choose a workflow specification language. Such a language has to be analyzed from two points of view. Firstly, there is the point of view of the concepts included in the language. This point of view can be associated to the syntax of the language and it has a deep impact on the usability of the language. Furthermore, this point of view also includes the semantics of the single elements in the language, which include its behavior and interaction with external elements.

Secondly, there is the point of view of the execution semantics of the language as a whole, that is the way in which various elements interact, the way in which they can be combined, and the way in which the run time state is represented. For instance, this aspect includes the order in which elements in a workflow specification are executed, or the way in which concurrency is handled in the language.

This section analyzes these two points of view using examples. First, we present five different workflow specification languages, and we highlight the differences in their characteristics and in their motivations. Afterwards, we present five models which have been widely used to describe the execution semantics of workflow languages. Finally, the concepts of expressive power and suitability are discussed and it is shown how they can be used to compare workflow languages and execution models.

2.2.1 Workflow specification languages

In this section we analyze five workflow specification languages, that present interesting differences. The main reason behind those differences, are the motivations that guided the design of those languages. The languages that are described are the following:

- **BPMN:** a standardized graphical notation for business processes that mainly targets communication among business experts, analysts and software developers.
- **WS-BPEL:** a standardized XML language for web-service based executable business processes, and web-service composition.
- **YAWL:** an academic workflow specification language with solid formal semantics.
- **Sedna:** a domain specific workflow language designed and implemented by physicists, on top of WS-BPEL. Sedna is used to model experiments in grid-based scientific applications. Although it is not a very well known language, Sedna is interesting for our research as a language used and designed by domain experts.
- **IMS-LD:** a specification for defining workflows in the e-learning context, using components and learning materials designed according to several complementing specifications.

BPMN

BPMN, or Business Process Modeling Notation, is a standard developed by the Object Management Group – OMG and the Business Process Management Initiative – BPMI in order to provide a notation to describe *business processes*. The main goals behind this standard are to provide a notation that is understandable for all business users (business analyst, technical developers, and the people that monitor and control the processes), and to provide a notation to visualize executable business languages [Obj09a]. BPMN 1.1 [Obj08] has been in use for a few years now, but an updated version, BPMN 2.0 [Obj09a], has been proposed and is currently in a beta stage.

The notation proposed for BPMN is mainly graphical, although also BPMN 2.0 proposes textual representations based on XMI and XSD. Figure 2.2 presents a sample BPMN diagram, which illustrates a few elements of the language. This diagram represents a workflow formed by a main process which includes a subprocess. The following are some of the main elements depicted in this figure:

- Tasks: they represent actions to be performed in the process and they are represented with rectangles that have rounded corners.
- Events: they represent special conditions that happen during the execution. For instance, they can represent the beginning of the process, its ending, an error, or the reception of a message. In the figure we have included a representative set of the available types of events, which are represented with circles and a variable icon inside.
- Gateways: gateways in BPMN are used to condition the control flow and to control the concurrent execution of various parts of a process. Gateways are represented with diamonds and the icon in them represents the way in which they handle concurrency.
- Swimlanes: swimlanes are the mean to differentiate which resource has to perform each action in a process. In the figure there are three swimlanes

which correspond to the three roles that participate in the process (Order System, Process Manager, and Testing System).

- Artifacts: although BPMN is not a data-flow, it offers a simple way to describe the data produced and consumed by each task. Nevertheless, the semantics of the execution does not depend on the flow of artifacts and it is only considered documentation.

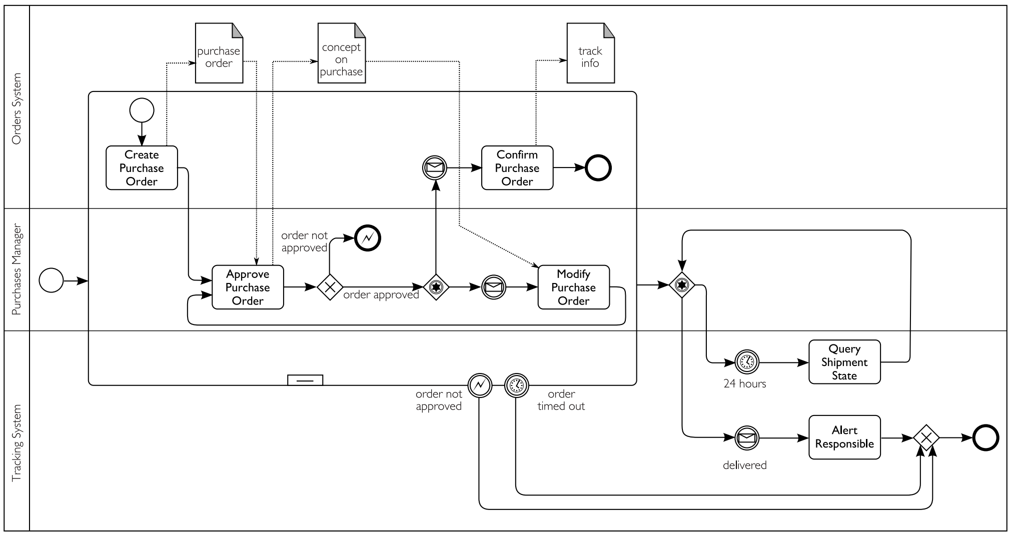


Figure 2.2 Sample BPMN diagram

As other OMG specifications, e.g. UML [Obj07a, Obj07b] or SPEM [Obj05], the BPMN 2.0 specification uses metamodels to introduce the elements of the language and the relations that can be established between them. The complete metamodel presented in the specification has more than 150 elements. This is new for version 2.0, as in version 1.1 the specification was more informal and also smaller. In comparison, the WS-BPEL 2.0 metamodel has less than 65 elements [wsp07], while YAWL has 14 [vdAtH06]. Therefore BPMN has been criticized for including too many elements, which are seldom used, and thus complicate the language [zMR08].

Since its first version, the main goal of BPMN has been to provide a communication mechanism. Thus, BPMN is used to communicate business processes between people with different skills and roles in an organization. Furthermore, BPMN is also a valid tool to represent and communicate processes that involve various organizations.

On top of that, BPMN has been used to define business processes with the explicit goal of enacting them. This was a problem in prior versions of BPMN because the semantics of the language were ambiguous and incomplete. Only a subset of BPMN had precise semantics, which were defined with transformations to WS-BPEL. Thus, many BPMN processes could be executed using WS-BPEL engines. BPMN 2.0 formalized the execution semantics for all elements [Obj09a], thus making it possible to build a BPMN engine which does not use WS-BPEL.

BPMN offers two mechanisms to extend the language and accommodate specific needs. On the one hand, BPMN's elements allow specialized markers to convey specialized information. For example, `Events` can be marked with icons that correspond to user specific extensions. The second mechanism is based on elements called `Artifacts`, which are user defined and are connected to other elements in the processes using `associations`. Nevertheless, `Artifacts` have a strong limitation: they cannot change the normal flows (i.e. Sequence or Message Flows) and they cannot alter the mappings to executable languages. As a result of these limitations, BPMN's extension mechanisms are useful for specializing the behavior of some of its elements, but are not enough to introduce new structures.

In spite of its limitations and drawbacks, BPMN is currently one of the most popular workflow languages available. According to the report in [Obj09b], there are at least 61 BPMN implementations reported to the OMG, which include editors, engines, monitoring applications and others.

WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL), also known as BPEL, is another standard to describe workflow definitions. It was proposed by OASIS and its main goal is to describe web-service based executable business processes [OAS05, KKM09]. While the motivations behind the design of BPMN are mostly related to the need of communicating users, in the case of BPEL the motivations are more related to the need to communicate and exchange information between companies and between machines. A further motivation behind BPEL's design is building complex services as a composition of existing ones.

In comparison to BPMN, BPEL is a low level language which is more suited for software systems than for business analysts and managers [Obj09a]. On the one hand, BPEL models are represented using a verbose XML based notation. Graphical representations are also used, but they are defined and provided by tool vendors, and they are not standardized. On the other hand, the concepts appearing in the BPEL language are of a lower level than the elements in BPMN and most of them refer to one of the following things:

- Web services consumption.
- Manipulation of XML data.
- Control flow elements (scopes, conditionals, loops).
- Exception handling.

From the point of view of the semantics, there are also important differences between BPEL and BPMN. While in BPMN the semantics are informally described (even in version 2.0), in BPEL these are more precise, and they have even been described with formalisms such as Petri nets [HSS05] or pi-calculus [LM07, Abo06].

The specification for BPEL discusses three possible kinds of extensions. The first kind involves the creation of new types of activities, which serve to introduce specialized behavior into new processes, and which is simply based on a single element: `<extensionActivity>`. The second kind is similarly used, and it serves to create new assign operations that permit new ways of managing data: `<extensionAssignOperation>`. Finally, the third kind of extensions involves the introduction of namespaces which can define additional attributes for existing elements, and additional elements. Nevertheless, extensions must not contradict the semantics defined by the BPEL specification, and this restriction greatly reduces the power of the extension mechanisms. Another problem with this kind of extensions is that, in the practice, it is not easy to modify the engines to make them work with the extended elements. One example of this is BPEL4PEOPLE [AAD$^+$07], the extension to BPEL that serves to manage human tasks. As reported in [HVD08], supporting this extension in an existing BPEL engine is not trivial, not because of the inherent complexity of BPEL4PEOPLE, but because of the difficulties of integrating it with the base engine.

Currently, BPEL is one of the most used workflow languages, and it has been applied in many different contexts, like business processes, scientific applications, computer aided engineering, and e-learning. This has been due to the characteristics of the language, and the widespread availability of tools to model, enact, and monitor BPEL processes.

YAWL

YAWL (Yet Another Workflow Language) [vdAtH06] is a workflow language that originated in the academia and has been mainly used for research purposes. YAWL offers a graphical syntax but it also has a formal definition. The semantics of the language are formally specified using Petri nets [RtHEvdA07]. Moreover, the semantics of YAWL are also formally specified using an extension to Petri nets known as EWF-Nets (extended workflow nets).

An example of the graphical syntax of YAWL can be seen in figure 2.3. This figure represents the same workflow that was previously used as an example of BPMN (figure 2.2), although it has a slightly different organization. With respect to BPMN there are various differences that can be seen immediately. The first one is that YAWL does not have anything similar to BPMN's events; their role is played by tasks such as those marked with a M that expect the reception of messages. Another one is that each process (or net in YAWL's terminology) has a single exit point. Finally, YAWL does not have something equivalent to BPMN's swimlanes or artifacts.

The designers of YAWL, which are also the original proponents of the control flow patterns [vdAtHKB03], designed the language with the explicit goal of supporting all these patterns. As a result, YAWL is the only widely known language that supports all the original control flow patterns[4]. On the other hand, YAWL also strives to be considered as an intermediate language that can be applied in a large number of contexts. Therefore, its elements are not particular to any specific domain.

YAWL does not offer mechanisms to extend the language by adding new elements or constructs, but it does offer the means to specialize the behavior of some kinds of elements. More specifically, the element of YAWL that is called `Decomposition` is a placeholder for concrete activities, and process developers are expected to implement decomposition specializations. On top of this, YAWL offers another mechanism of extension which is based on `Worklets` and which allows for the late selection of the activities of execute in a process.

[4]There is a pattern not supported in YAWL (implicit termination) because the authors of the language consider that pattern to be a potential source of errors. Nevertheless, the pattern can be supported by applying a simple transformation in the processes, to connect the dangling tasks to the end condition of the YAWL process.

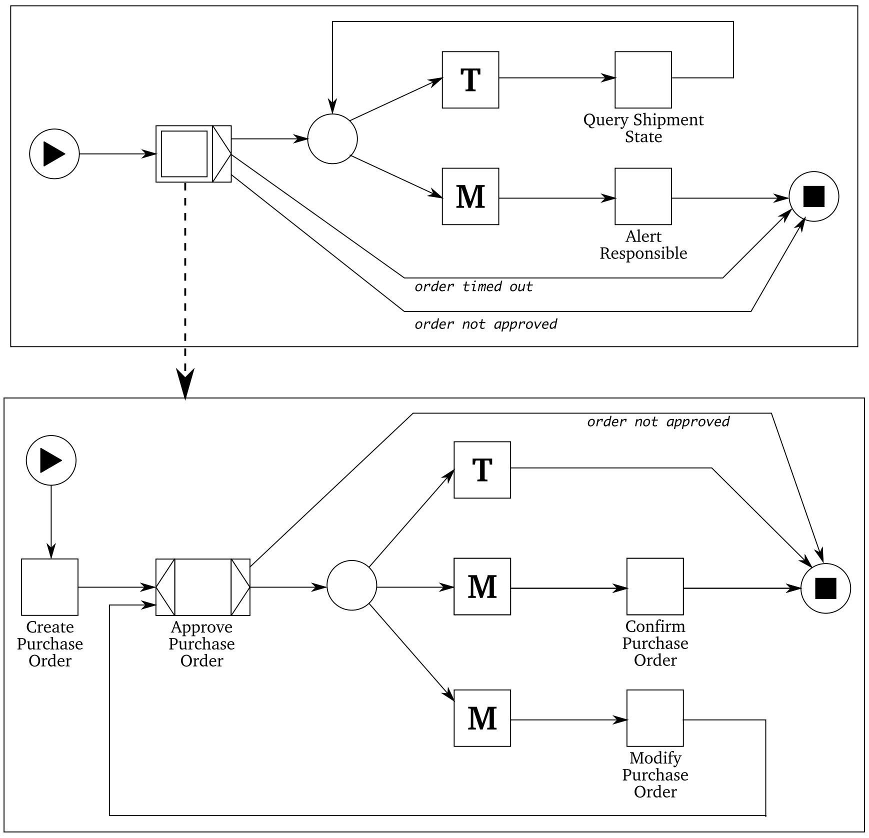


Figure 2.3 A sample YAWL process. The net on the upper part is the top level net, while the one on the lower part of the image is the sub-net.

Another reason that makes YAWL important for the academic community is that it has been used as a test bench to develop analysis tools for workflows. These have been mainly based on applying techniques developed for Petri nets analysis or for graph-analysis. Woflan [vdA99, VvdA00] is one of the resulting tools for workflow analysis, and it can be used to discover flaws in the design of business processes.

Sedna

We are now going to analyze the main characteristics of a language that has been designed for a specific domain. The term Sedna refers both to the workflow specification language based on WS-BPEL and to a visual modeling environment to define workflows in the context of scientific applications [WEB$^+$07].

The main motivation behind Sedna's design is offering the means to successfully apply BPEL to scientific applications. There are two reasons for this. On the one hand, BPEL has enactment environments that offer the scalability and reliability required in scientific applications. On the other hand, the abstractions used in BPEL lack the expressiveness needed for scientific applications.

Sedna's design also takes into account the most important characteristics of scientific workflows, which set them apart from business workflows. These characteristics are the following:

1. The large scale of the processes, which may make thousands of invocations

to many different services.

2. The requirement of executing in parallel a large number of very similar subworkflows.
3. The massive amounts of data produced and processed by each process.
4. The short life of the processes, which are frequently changed or adapted as experiments are refined.

Given this situation, the goal of Sedna was to provide a modeling language and environment to build scientific workflows, and also to take advantage of the features offered by BPEL. To achieve this, they built an environment where the complexities of BPEL are hidden, and created a visual extension to WS-BPEL which incorporates concepts that are relevant to scientific workflows. Behind the scenes, the environment transforms Sedna workflows into standard BPEL processes and creates deployment descriptors to run these languages on some of the most popular BPEL engines.

With respect to standard BPEL, Sedna introduces the following four extensions:

1. **Indexed flows.** Indexed flows are containers where activities can be placed for parallel execution. Therefore, indexed flows can be used to model the massively parallel execution of subworkflows.
2. **Hierarchical compositions.** Since scientific workflows are much more complex than business processes, some elements to reduce the complexity and simplify the development were added. By introducing the hierarchical composition of workflows, Sedna's developers gained the two things: on the one hand, this allows reusing existing workflows; on the other hand, it reduces the complexity of large workflows.
3. **Plug-ins.** In Sedna, *plug-ins* encapsulate parameterizable sequences of activities using Java classes. When plug-ins are included in a workflow, they look like basic BPEL activities. However, when the standard BPEL process is generated, the included plug-ins are run and modify the generated structure. For instance, plug-ins can be used to easily introduce complex sequences of activities that process the data produced by an experiment.
4. **Macros.** Macros are inlined BPEL activities that are expanded when a Sedna workflow is converted into a standard BPEL process. Macros, as well as plug-ins, are intended to be shared and reused by many developers across multiple experiments.

The final important point about Sedna is related to tool support. Besides the modeling environment for the new language, Sedna's developers also had to build an execution environment that permits the interaction with the workflow instances. This environment plays a crucial role in the success of Sedna, as it has several responsibilities. On the one hand, it has to enable the quick modification of the scientific workflows and it has to validate their correctness with respect to certain criteria. Furthermore, it offers the means to monitor and debug the enactment of the processes. However, all this must be realized while hiding the complexity of BPEL and its inherent technical details.

As a domain specific workflow language, the development of Sedna shows what language developers must do in order to have full support for their languages. First of all, they have to design the language: in the case of Sedna, this required the definition of a visual representation, and the definition of the required BPEL extensions.

Then, it is necessary to develop the tools to use the language, which normally are at least an editor, and an engine. In this case, they developed a fully-fledged visual modeling environment, and converted the scientific workflows into BPEL processes. In this way, they did not have to create a new engine for the language.

Finally, they had to consider additional requirements (validation, debugging, monitoring) and included support for them in the tools. In this case, the conversion of Sedna to BPEL created some additional problems to implement these functionalities, in comparison to using a *native* Sedna engine.

IMS-LD

The IMS Learning Design Information Model, or IMS-LD, is a specification that integrates a number of other specifications in the context of e-learning [IMS03b]. Using IMS-LD, it is possible to model *learnflows*, which are equivalent to workflows in the e-learning context [MnCVC07]. Basically, a learnflow is a sequence of activities which have a pedagogical goal and are performed by learners and by a supporting staff, using a number of learning objects.

When the IMS-LD specification was created, it provided a framework to integrate several existing specifications, which targeted very different aspects of the e-learning problem. For instance, the IMS Question and Test Interoperability specification defines how evaluation applications can be specified and packaged to be included in a Learning Design; instead, the IMS Learner Information Package specification provides mechanisms to model and store the information about learners, which may be modified during the execution of a Learning Design. The IMS-LD specification does not offer itself extension mechanisms, but there is flexibility in its support for multiple implementation of the complementary specifcations.

Among the requirements of the IMS-LD specification, there were several related to the execution of the Learning Designs. In the first place, it had to allow the participation and interaction of both learners and staff members. Furthermore, learners could participate as single users, or as part of groups. Finally, a very important requirement was supporting the interoperability between Learning Designs and other applications. On the other hand, from the language point of view, IMS-LD had to be powerful and flexible. It had to support any pedagogical model, and it had to allow the personalization and adaptation of the units of learning.

Given this context, it can be seen that the design of IMS-LD faced a number of difficult restrictions imposed by the domain. The elements of IMS-LD, which are depicted in figure 2.4, are organized according to a theatrical metaphor. A *play* specifies an actual learning design, the teaching-learning process. Each play, which is composed by a sequence of *acts*, specifies which *roles* perform which *activities*. The *persons* that participate in a play have to assume a role as *learners* or as *staff*. The conceptual model includes also a number of elements that are exclusive to the e-learning context, such as learning objectives, prerequisites, and learning objects.

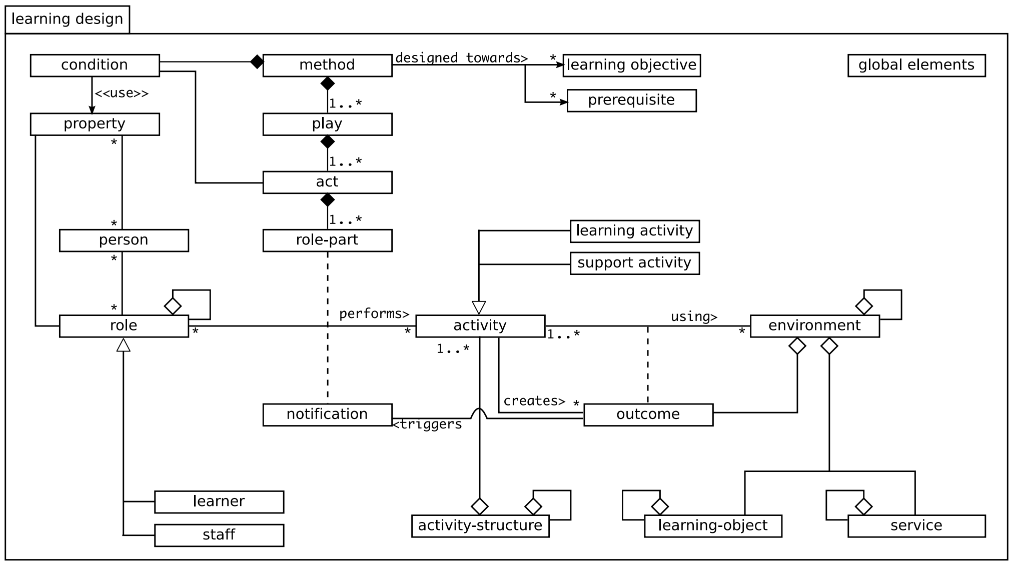


Figure 2.4 Conceptual model of IMS-LD [IMS03b]

Besides specifying the conceptual model, the IMS-LD specification also defines three compliance levels. Implementations that are compliant with level A of the specification are only required to support the basic part of the specification, which defines static and fixed learning designs, and does not handle properties or personalizations. Compliance level B includes level A and adds *conditions* and *properties*. Using those, different learners may follow different learning sequences. Finally, compliance level C adds *notifications* which can be used to make new activities available for a role to perform. Notifications introduce dynamic adaptation into learning designs.

In spite of the powerful characteristics of IMS-LD, there are only a few tools available which support the specification. Furthermore, most of them have compliance only with the level A of the specification. This is probably due to the complexity of the requirements imposed by the IMS-LD specification with respect to interoperability, the support of the other specifications, and the dynamic adaptation of the learnflows. Since these requirements are inherent to the domain and are likely to appear in any other language to describe learnflows.

Summary about the workflow specification languages

This section has presented a sample of the spectrum of workflow specification languages. The languages presented differ in several aspects, and this contributes to showing the diversity of languages available. The first characteristics to compare these languages is their relative genericity or the domain that they target. We presented three languages that can be applied in a wide number of domains (BPMN, BPEL, and YAWL), and thus can be considered generic. Conversely, we also presented two languages that are restricted to specific domains (Sedna and IMS-LD).

Another difference between the presented languages concerns the number of elements in them. BPMN is, by far, the biggest of the languages presented, and it includes elements to describe many workflow concerns. On the contrary, YAWL mostly focuses on the control flow[5]. IMS-LD has also the potential of managing a large number of concepts. However, these concepts are modularized across several specifications, and are not mandatory.

The languages presented also differ on the extension mechanisms that they provide, but at least all of them are capable of supporting specializations at the activity level. Extensions with more ample impact to the languages' structures require bigger efforts to be supported, and this explains why some of these languages put limitations to the kinds of extensions allowed. Furthermore, we can see that BPMN and BPEL restrict the extensions to prevent changes to the basic semantics of the languages, and thus guarantee basic compatibility between engines.

Finally, a further criterion for comparing these languages is the way in which their semantics are specified. In YAWL, these are formally declared. In the case of BPEL, the semantics have been formalized as well, although this is not included in the specification. The semantics of Sedna are not explicitly specified, although the mapping to BPEL solves the problem. On the other hand, the semantics of BPMN and of IMS-LD are not formally defined and thus are subject to interpretation by the implementers.

2.2.2 Execution models

The execution model of a workflow specification language is an abstract model of its operational semantics. In the literature, it is also known as the *workflow model* [GHS95], the *internal process definition* [Hol04], or as the formalism to define the control aspect of a workflow [KAR06]. The execution model determines three aspects of the language execution.

In the first place, there is the *order* in which actions defined in a workflow specification have to be performed. In execution models that do not support concurrency, there has to be a policy to serialize the actions in a workflow. Otherwise, the execution model defines which actions can be run in parallel, and how their execution must be coordinated.

A further responsibility of an execution model is establishing when the elements in a workflow definition can *interact with their environment.* This includes handling interactions coming from outside of the workflow instance or even outside of the engine. Some execution models also consider the *management of expected or unexpected errors* (i.e. exception handling).

Finally, the execution model also defines how the *run time state* of a workflow is represented and how it is modified. For instance, an execution model specifies how each operation modifies the state of a running instance of a workflow.

On the other hand, there are also aspects of a language's semantics that are usually not considered to be part of its execution model. These aspects include, for example, the fine grained details about the interaction with external applications (e.g. the communication protocols) and about the mechanisms to process and store data. By leaving these details out of the description of the execution models, they are kept from becoming excessively complicated. This facilitates the analysis of the languages, and the construction of tools based on these execution models.

[5]The YAWL editor also includes mechanisms to specify resource assignments, and also time restrictions. However, it is not clear where to draw the line between YAWL as a language and YAWL as a tool.

There are three main uses for the execution model of a language. In the first place, an execution model is necessary to *enact the processes* with a workflow engine [vdAtHW03]. In particular, this requires keeping updated the state of the process and executing the actions, by following the operational semantics established in the execution model. A similar and complementary use for the execution models is to *simulate the execution of process definitions*. This is usually done when the processes are being designed, and it is a technique for the detection of problems in process specifications.

Finally, the information about an execution model is also necessary to *analyze workflow definitions*. This is done especially with execution models that are formally specified, and it has been applied in complex contexts, such as the SAP reference processes [vDJVVvdA07].

Below we present some of the types of execution models that are more frequently used with workflow specification languages.

Petri nets

Petri nets are a formalism to model concurrent, distributed systems, which is based on a small set of simple concepts and on formally defined state-transition rules [GV01]. This formalism has been applied to different domains, its advantages and disadvantages have been thoroughly studied, and many tools have been developed to work with them in both specific and generic domains. Petri nets have been used to describe the execution model of workflow specification languages because of several reasons.

The first one is that the semantics of Petri nets are formally designed and thus allow for the precise definition of the core aspects of the workflow specification languages' semantics. One example of this is the YAWL language, whose semantics has been defined first using an evolution of Petri nets (see Extended Workflow Nets below), and then using Colored Petri Nets. They call this alternative definition newYAWL [RtHEvdA07].

A second reason is the availability of tools and techniques to analyze or simulate Petri nets. Most of these have not been initially developed for workflow specification languages, but they have been successfully adapted. YAWL is also an example of this. On the one hand, the core of newYAWL's engine is a Petri net interpreter. On the other hand, Petri nets have been used to analyze the structure of YAWL processes [vdA99, VvdA00]. Similar things have been achieved in other workflow specification languages: BizAgi is a Petri nets-based engine for BPMN [Biz10]; and LoLA is a tool to verify WS-BPEL specifications that is based on representing them as Petri nets [HSS05].

Another reason for using Petri nets to describe the semantics of workflow specification languages is that it offers a relatively simple, but very powerful, model. In its basic version it includes only four elements (places, transitions, edges, and markings) which are generic. In comparison, other formalisms and execution models use more specific elements and have more complex semantics. As a result, these other formalisms are more difficult to be used with several different languages, and cannot be applied as easily to diverse domains.

Finally, Petri nets offer a very powerful model of concurrency based on disjoint locality [GV01]. Therefore, they are suited to represent workflows where actions are executed in parallel, and concurrency issues are only considered in the points where synchronization should happen.

The application of Petri nets to workflows has been studied, and two main alternatives have been identified to map workflow concepts into nets [GV01]. These alternatives emphasize different aspects of the processes. In the most common mapping strategy, there is a transition for each task, and control flow is determined by edges and shared nodes. Therefore, the state of a workflow is determined by the current marking. The exact meaning of a transition firing has to be specified in each case, because it could signal the start or the end of a task execution. To solve this issue, a second strategy has been proposed, where each task is represented by a transition followed by a node followed by another transition. Thus, the first transition marks the beginning of the task, the second transition marks its end, and a token on the node signals that the activity is in execution.

These two alternatives are depicted in figure 2.5. Part *a* of the figure shows a fragment of a BPMN process. Then, part *b* shows how this fragment could be represented in a Petri net if the first mapping strategy was applied. Finally, part *c* shows the result of applying the second strategy. In this figure, B_i marks the beginning of task B, while B_f marks its end. A third strategy that has been proposed represents tasks with nodes, but apparently this has not been implemented [EW01].

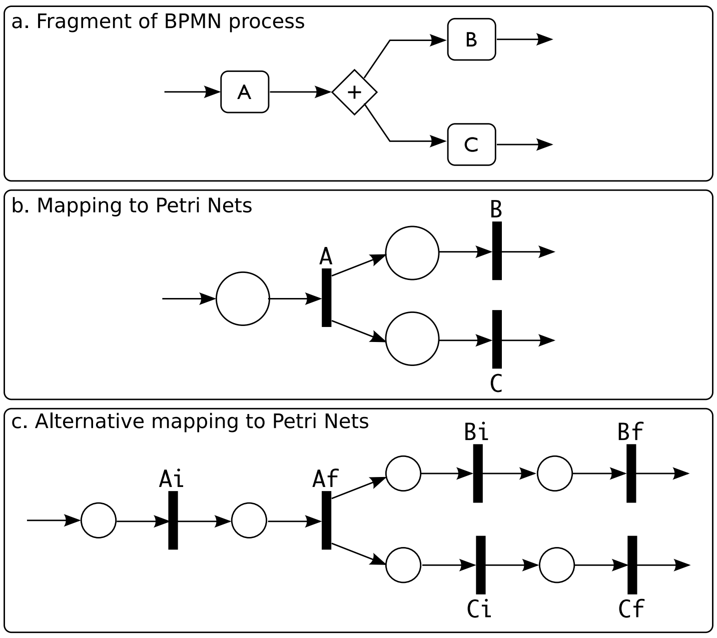

a. Fragment of BPMN process
A
+
B
C
b. Mapping to Petri Nets
A
B
C
c. Alternative mapping to Petri Nets
Ai
Af
Bi
Bf
Ci
Cf

Figure 2.5 A sample BPMN and two possible mappings to Petri nets

Using Petri nets with workflow specification languages has also some problems. On the one hand, the simplicity of the model can make it necessary to use a large number of elements, or complicated structures, to represent even simple workflows. This occurs because each element in a workflow specification language has usually a complex semantics that has to be represented with several nodes, transitions and arcs. Eshuis and Wieringa have also identified other characteristics of Petri nets that make them inadequate to represent workflows [EW01]. The main one is that Petri nets are usually non reactive, while workflows typically depend strongly on external interactions.

Extended Workflow Nets

Because of the problem of relative simplicity in Petri nets, some languages use execution models inspired on Petri nets. One well known example are *Extended Workflow Nets* (EWF-nets), which are the real execution model for YAWL. In [vdAtH06], the authors of YAWL formalize the structure and behavior of an EWF-net.

With respect to Petri nets, the most important characteristics of EWF-nets are the assignment of identifiers to net instances, the inclusion of conditions associated to the edges, and the possibility to remove tokens from places. These characteristics enabled the inclusion in YAWL of concepts such as *cancelation regions* and *composite tasks*, without complicating the models. In spite of all these additional elements, the designers of EWF-nets claim that they can be converted into Petri nets [RtHEvdA07].

Transition based systems

Transition based systems are another derivation of Petri nets which are frequently used as execution models for a workflow language [Hol04]. The main characteristic of these systems is introducing elements with more complex semantics than those found in Petri nets. Therefore, in these systems transition firing conditions are subject to more complex rules.

Besides the nodes, transitions, and edges of Petri nets, transition based systems usually include complex joins and splits. Furthermore, they typically support conditions associated to the transitions, and thus make routing decisions more complicated. In order to be able to use these conditions, there has to be a language available to express them.

Compared to using Petri nets, the most important benefit of using a transition based system is the reduction in the number of elements in the models. This is explained because each element in a model has a semantic that can be equated to that of a Petri net with several nodes and transitions. The tradeoff in this case is in the additional complexity of each element and of the execution rules. Therefore, transition based systems are more complicated to analyze, and this explains why most tools to analyze workflows are based on Petri nets.

A further disadvantage of transition based system is that they are less likely to be reused. Since they include higher level concepts, the supported workflow languages need to be compatible with those concepts, which are less general than the concepts included in Petri nets.

Block structured decompositions

Another common type of execution model is based on the concept known as ‘block structured decomposition’. In the languages that use this model, process specifications are divided in blocks that can be sequential or parallel. Concurrency is restricted so no two parts can be executed at the same time, unless they are parallel and some conditions are met. Furthermore, the decomposition can be hierarchical, and the identified blocks can be typed and have different behaviors. As a consequence of the hierarchical structure, languages based on this execution model cannot handle arbitrary loops, or have unstructured splits and joins [Hol04, vdAtHKB03, RtHvdAM06].

BPEL is one of the best known languages that uses a block structured decomposition. In this language, blocks are known as scopes and, depending on their type, they have different structures and behaviors. For instance, `Sequence Flows` execute in sequence a collection of `Activities`. Another example are `Whiles`, which execute repeatedly a `Sequence Flow`, until some associated condition is false. Another workflow system based on this execution model is the Windows Workflow Foundation [Mic07].

Rule based execution

The final type of execution model we discuss is based on the application of ECA rules to workflows [MD89]. With this execution model it is not necessary to establish explicit dependencies between activities in workflows. Instead, these dependencies are replaced with triggers, and conditional flows are replaced with conditions. Finally, the action part of the ECA rules define the specific activities to execute when the rule is evaluated.

As an example, consider the BPMN process fragment depicted in figure 2.6. In this process, task 'B' and task 'C' are executed depending on the result of evaluating the functions `f( )` and `g( )`, which are not necessarily mutually exclusive. If this process is executed using a rule based execution model, the two flows that lead to 'B' and 'C' can be represented with two rules:

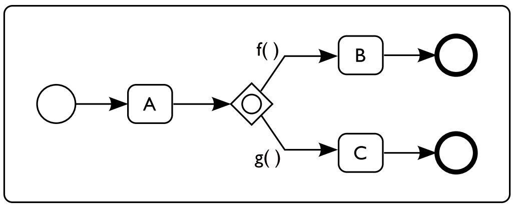


Figure 2.6 A simple BPMN process

Table 2.1 Representation as ECA rules of the process in figure 2.6

	Rule 1	**Rule 2**
Event	Completion of activity 'A'	Completion of activity 'A'
Condition	`f( )`	`g( )`
Action	Execute 'B'	Execute 'C'

Two workflow systems based on this execution model are EWMS [WLC$^+$05], and JOpera [Pau09]. In section 2.4.4 we present more details about JOpera.

2.2.3 Expressive power and suitability

The two previous sections presented examples of workflow languages and execution models which demonstrate the variance that can be found in this context. Generally speaking, it is positive to have so many alternatives because it makes more likely to find a very adequate solution for each problem. However, this also creates a burden for those in charge of selecting the language to adopt in a project. Furthermore, there is also the alternative of developing a new, ad hoc, workflow definition language to solve the specific problems that are faced.

In many respects, this problem is similar to that of selecting a programming language. In order to take such a decision it is necessary to analyze the available languages from several points of view. On the one hand, there are practical aspects to consider, like the availability of tools to use with each language. It is also important to analyze the availability of documentation and support. In short, these factors are important because they have a big impact on how efficient are users of the languages.

On the other hand, languages should also be analyzed and compared with respect to their inherent characteristics. However, finding the criteria for this comparison is a complicated problem. In his dissertation, Bartosz Kiepuszewski proposed the idea of using two complementing metrics to tackle this problem: expressive power, and suitability [Kie03].

Expressive power

Expressive power refers to workflows that can, or cannot be modeled with a given set of modeling constructs. It is an *objective* criterion, because it is possible to demonstrate whether some structure is supported or not[6]. For instance, Kiepuszewski demonstrated that *Standard Workflow Models* have less expressive power than free-choice Petri nets [Kie03].

Expressive power must not be confused with *expressiveness*. In the context of programming languages, when a language ‘a’ is said to be more expressive than language ‘b’, it usually means that it is more succinct or has less *syntactic sugar* [Fel90]. Felleisen proposed a framework to formalize the measurement of expressiveness in [Fel90], but still this measure is frequently evaluated in a subjective and imprecise way. On the contrary, the measurement of expressive power proposed by Kiepuszewski is not normally applied to general purpose programming languages because these languages are usually universal [Fel90]; therefore, comparing the sets of computable functions is frequently useless.

In the context of this dissertation, expressive power is relevant because it allows to compare execution models with respect to the requirements of a certain domain or of a workflow language.

Suitability

The second metric used by Kiepuszewski is *suitability*, and it complements expressive power. Loosely speaking, the term suitability refers to how ‘straightforward’ or how ‘naturally’ a modeling problem is solved with a given language. Suitability also refers to the match between available constructs in the modeling language and the concepts in the application domain [Kie03]. Therefore, suitability is a subjective notion, which depends not only on the workflow language but also on the domain where it is applied.

In previous work, including Kiepuszewski’s, suitability has been ‘measured’ using various kinds of workflow patterns [vdAtHKB03, RtHvdAM06, RtHEvdA04, RvtE05]. In these works, the evaluation usually involves an expert on the language that models each proposed pattern. Afterwards, someone *evaluates* those

[6]Nevertheless, this demonstration is only possible if the the modeling constructs are formally defined.

models, judging them as suitable (+), somewhat suitable (+/-), or unsuitable (-). This is done based on entirely subjective criteria, and has been a source of debate. Nevertheless, this kind of evaluation has been applied to a large number of languages and standards (including BPEL, BPMN, XPDL, UML, EPCs and YAWL [Word, vdAtHKB03, WvdADH03, WvdAD$^+$06]), commercial systems [Worb, vdAtHKB03, RtHvdAM06], and open source systems [Worc].

These evaluations are not necessarily *fair*. On the one hand, they evaluate or compare the languages using abstract scenarios described in patterns. Thus, the evaluation is limited only to very specific details and it is not possible to extract conclusions about issues like the suitability with respect to composed patterns. On the other hand, patterns' definitions are usually informal, and thus ambiguous and subject to interpretation. Because of this, it is not possible to decide if two models built with different languages are really modeling the same pattern. There have been a few attempts to formalize the control flow patterns [WG07, PW05, GT08], but these formalizations have not been widely adopted.

In spite of these deficiencies, suitability is a valuable metric to aid in the selection of a workflow system. This is because suitability usually has a direct relation with other important characteristics such as maintainability, understandability, and usability.

2.3 Workflow modularization

Typically, workflow definitions are built monolithically. This means that every workflow is specified in a single artifact that cannot be modularized. The only widespread exception to this are sub-workflows, which represent fragments of a bigger workflow. However, this mechanism is not powerful and general enough to solve all the problems related to the lack of modularization techniques for workflow definitions.

One of the issues associated with this situation is that workflow specifications are much larger than they would be if there were adequate modularization mechanisms. This complicates the specifications and makes them harder to understand, analyze, and maintain [BVJ$^+$06]. Furthermore, reusability is hindered, as it is restricted to reusing entire sub-workflows [Jab94, CM06].

The lack of modularization mechanisms for workflows also introduces scattering and tangling problems [CM06, BVJ$^+$06]. Every workflow definition involves the description of multiple concerns of the represented process, which include the activities to perform, the people involved, the produced and consumed data, and also technical concerns such as distribution, data validation, and security. Since the only available decomposition mechanism (sub-workflows) modularizes the workflows according to the control flow, the other concerns are scattered across the workflow definition, and tangled one to another. In consequence, it is very difficult to analyze or modify only one concern in isolation, and it is also complex to introduce new concerns into an existent workflow.

Finally, evolution and adaptation of workflows is another aspect where more powerful modularization mechanisms are necessary. Currently, "workflow languages do not support a modular expression of changes as first-class entities" [CM06]. Therefore, changes are unstructured, introduced ad hoc, and thus are difficult to track or revert.

Only a handful of projects have tackled these issues and proposed alternatives to modularize workflow definitions. In the following sections we briefly introduce those projects and the strategies they have followed. In the first place, we discuss the identification and usage of perspectives on business processes, and then we discuss the application of aspect-oriented concepts and techniques in the context of workflows.

2.3.1 Business process perspectives

The literature on business process modeling has consistently identified some perspectives that can be used to analyze and model each process. These perspectives can also be seen as dimensions that business process modelers should take into account and use as reference, but traditionally they have had a more important role in the documentation than in the actual implementation of the processes [Jab94].

Below we describe the perspectives that appear more commonly in business process literature. They can also be generalized to most workflows.

- Functional Perspective.

  This perspective considers the goals and the actual activities (or tasks) performed in a business process. In all the other perspectives, ‘tasks’ are concepts that perform abstract or unknown actions. For instance, in the functional perspective it is established if a task is manual and performed by a user, or if it is automatic. In the first case, the perspective includes all the information to establish what the user must do. In the second case, the perspective includes all the information to execute the task, which may involve transforming data, invoking other applications, or consuming services.

- Behavioral Perspective.

  This perspective considers the interdependencies and interrelationships between tasks in a process. A commonly used mechanism to establish these interdependencies are control flows, that determine which activity can be executed at each point. There are two basic types of control flows [Jab94]. The first type are *prescriptive* (or sequential) control flows, that establish at each step which are the following tasks to execute. The second type are *descriptive* control flows, which establish what must happen before each task can be executed.

- Organizational Perspective.

  This perspective describes the structure of the organization in which the workflow is executed. This includes describing the characteristics or abilities of the users involved, and also the characteristics of other resources that can be used, such as machines. This perspective also describes rules to assign tasks to users.

- Informational Perspective.

  This perspective describes the information consumed and produced by a workflow. This includes how documents flow between tasks (i.e. the data flow) and also how tasks have dependencies based on the data they need.

Workflow perspectives have been used to modularize workflow definitions in a number of tools. MOBILE is one such tool [Jab94]. In MOBILE a workflow specification is composed of several aspects[7] that correspond to different workflow perspectives. In MOBILE, they have favored the four perspectives that were previously discussed, but it is possible to add application-specific aspects whenever they are needed. A further important characteristic of MOBILE is that for every perspective a different notation is used. In consequence, these notations can be very suitable.

Besides extendibility, the other important capability offered by MOBILE is reusability. This is supported with the MOBILE *built time architecture* [Jab94], which uses a repository as a library to store reusable parts of a multitude of workflow definitions. By offering this, the process of building a new workflow definition becomes more similar to a configuration process.

The other part of MOBILE's implementation is the *run time architecture*, which consists of two main blocks. On the one hand, there is the MOBILE Kernel, which interprets workflow definitions stored in the repository. On the other hand, there are various *servers*, which provide the functionalities for the different perspectives controlled by the MOBILE Kernel. Therefore, to support a new perspective it is necessary to first implement an adequate server, and then to configure it adequately so that the kernel can use it.

AMFIBIA is another proposal to achieve workflow modularization at the design and the execution level. Although AMFIBIA's documentation uses the terminology 'aspect' to refer to the parts of a workflow definition, their approach is closer to MOBILE than to the aspect-oriented workflow systems that we describe in the following section.

AMFIBIA is a metamodel that formalizes the essential aspects of business process modeling using an approach based on modularization [KAR06]. There are three main ideas behind the design of this approach. Firstly, AMFIBIA is formalism independent, meaning that the proposed metamodel can be adapted to many different formalisms. Furthermore, each aspect can have its own devoted formalism or notation. Secondly, AMFIBIA proposes a set of basic aspects, but it is not necessarily limited to those. New ones can be introduced as needed, and AMFIBIA guarantees that there is no bias towards any particular aspect. Finally, AMFIBIA does not aim to only describe simple relations or equivalences between elements occurring in different perspectives. Instead, AMFIBIA also captures the interactions between those elements, and uses that information to ensure that the execution semantics of the workflow is complete.

These ideas are implemented in AMFIBIA using a core metamodel formed with the elements that are common to all aspects. These elements are shown in figure 2.7. In addition, for each aspect there is another metamodel, which is formalism independent. The elements of those aspect specific metamodels are related to the elements of the core metamodel. Finally, several formalism dependent models can be related to every aspect specific metamodel.

The behavior of the elements in each aspect specific metamodel is determined using automata that vary for each aspect. The elements in the core metamodel have some basic automata, and the elements in the other metamodels are synchronized using an event based mechanism. Because of this, new aspects can be

[7]In MOBILE literature, they use the term *aspect*, but it does not have the exact same meaning as in AOP literature.

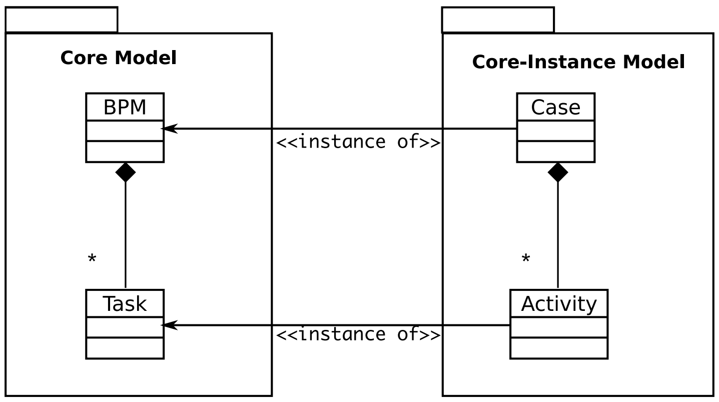


Figure 2.7 The core of the AMFIBIA model [KAR06]

integrated without much problems.

The two discussed approaches, MOBILE and AMFIBIA, show the main advantages of modularizaring workflow definitions using workflow perspectives. In the first place, the selected perspectives are usually well established and accepted by the community. Therefore, the decomposition criteria are well understood by most users, and the modularization is natural to them. Secondly, the two presented approaches are capable of managing perspective-specific languages. This is an advantage from the point of view of the suitability of the languages. Nevertheless, this can only work as long as the relations and interactions between languages and between perspectives are clearly defined. Unfortunately, part of the literature about workflows and workflow perspectives does not specify those relationships clearly enough to implement them.

2.3.2 Aspect orientation in workflow systems

Aspect-oriented software development (AOSD) techniques have as goal a better separation of concerns so that adding, modifying or removing such concerns has a reduced impact on the rest of the system [BVJ$^+$06]. *Aspect-oriented workflow systems* propose to apply AOSD concepts and strategies in the workflow context to solve the lack of modularization. These systems adapt and apply AOSD concepts such as aspects, join points, pointcuts, and advices [CM06].

AO4BPEL is an aspect oriented extension to BPEL that serves to modularize workflow definitions using a low level pointcut language based on XPath [CM06]. Using this language it is possible to locate the points in a workflow definition where the advices have to be woven. Furthermore, these advices must also be specified using BPEL.

Although AO4BPEL is based on BPEL, it is not compatible with standard BPEL engines. AO4BPEL introduces a feature that most BPEL engines do not support: dynamicity. Therefore, using AO4BPEL it is possible to introduce changes into the process definitions even at run time. To do so, changes are encapsulated as concerns and they are dynamically woven to the original processes.

Padus is another aspect oriented extension to BPEL which also intents to provide a better separation of concerns [BVJ$^+$06]. There are many differences between the approaches followed in Padus and AO4BPEL, and they have important consequences on the offered features. In the first place, Padus uses a static approach, where advices are statically woven to the main process. Since the result of this weaving process is a valid BPEL process, a standard BPEL engine can be used. The downside of this is that Padus does not support dynamic adaptation of the processes. On the other hand, the approach followed by Padus does not introduce any performance overhead, while in AO4BPEL it is possible to have performance problems related to the application of aspects.

Another important difference is due to the used pointcut languages. While AO4BPEL uses a language based on XPath, Padus uses a very expressive logic based programming language. In consequence, expressions are more easily read and it is possible to construct high level composition primitives. There is also a difference in the weaving strategies allowed. Besides the normal *before*, *after*, and *around*, Padus also allows more complicated weaving strategies.

Finally, AspectViewpoint is a domain specific aspect language for workflow models [Cor07]. It is based on model-driven engineering ideas, but it is mainly targeted at BPMN. With respect to the previously discussed approaches, the main difference of AspectViewpoint is to support the definition of viewpoints using control flow patterns [vdAtHKB03, RtHvdAM06].

With respect to the usage of perspectives for modularization, applying aspect oriented techniques into workflows has one main advantage. This advantage is an increased flexibility, because concerns can be freely selected or designed for each particular application. Therefore, more specific concerns, such as billing, can be selected. On the other hand, these approaches also have some important limitations. First of all, there is the fact that every concern has to be specified with the same language. In AO4BPEL and Padus this language is BPEL, while in AspectViewpoint it is BPMN. From the point of view of suitability this is not desirable, because these languages may not be adequate for other concerns. This is also related to the second limitation of those approaches, which is the asymmetric composition. As a consequence of this, the language used in those approaches is that of the *central* concern, i.e. the control flow.

2.4 Workflow kernels and intermediate languages

The goal of this section is to describe the most relevant characteristics of a number of systems that can be used as a base to build workflow engines. The systems described in this section were selected because they show the wide spectrum of strategies that have been applied to this end. The main characteristic shared by all these systems is that they offer support for the basic workflow functionalities. Thanks to this, they alleviate developers from re-implementing those. For the rest, they all have different goals and offer different architectures.

One criteria that differentiates these systems is how they are used in a software solution. Some of them, are intended to be used as the central element of workflow management systems. Instead, others are intended to be integrated into applications where workflow functionalities are needed, but are not the central requirement.

Another difference between these approaches is how they allow the interaction with other applications. In some cases, they offer object oriented APIs. In others, the interaction requires the usage of an intermediate workflow language or can also happen through data stored in a database.

The rest of the section is structured as follows. First, a few systems based on intermediate workflow languages and models is presented. Then, the main characteristics of Opera, a system based on databases is presented. Section 2.4.3 presents the main characteristics of two object oriented workflow kernels, namely Micro-Workflow and Lewfe. Finally, section 2.4.4 presents JOpera and discusses its usage of ECA rules and code generation.

2.4.1 Intermediate workflow languages and models

Intermediate languages and models are base formalisms used to support other higher level formalisms. The general idea is to only implement what is needed to support the execution of these base models, and to *transform* higher level languages into them. This is very similar to the strategy that we previously discussed to execute BPMN specifications on top of BPEL engines.

Intermediate languages take different forms, but they share two main characteristics. On the one hand, they cannot be too specific, or the high level languages supported would be severely limited. On the other hand, they need to have a high expressive power to support the execution semantics of the high level languages.

This leads to two main issues of intermediate languages [FCS04]. The first one is knowing if the formalisms selected are expressive enough to support the high level languages. The second one, is having enough flexibility to adapt those formalisms to new requirements. Unfortunately, these two questions depend on unknown requirements and cannot be answered with precision.

"Workflow Kernel" and "Workflow Virtual Machine"

Ferreira and Pinto Ferreira proposed in [FF04] an architecture to build workflow engines that use an intermediate model based on Petri nets. Their work was mainly motivated by the frequent need to redevelop common workflow functionality because of a poor support for reusability, and their proposal was a reusable and embeddable system that abstracts common workflow functionalities. They call this system the "Workflow Kernel", and it prevents the repeated implementation of general workflow features.

The central element of their Workflow Kernel proposal is a model based on Petri nets that is extended with the concepts of *actions* and *events*: an action is something that has to be executed when a token is received in a specific node; an event is something that triggers a transition (see figure 2.8). They selected Petri nets because they are regarded as capable of making the Kernel independent of the workflow language, and because of their formal semantics that make it possible to do process analysis and verification [FF04].

In this extended Petri nets model, all workflow tasks are regarded as arbitrary actions, which emit events to signal task completion, failure or timeout. Furthermore, these events trigger transitions in the Petri net, and thus they make the process execution proceed.

An extended version of this event based mechanism is used to augment the functionalities of the basic workflow engine that is at the base of the Workflow Kernel. For this, they defined some additional events that can be produced, and defined an interface (`INotifySink`) that must be implemented by any element that wants to interact with the workflow engine. The additional events include changes to the nets, changes to elements' attributes, triggering of actions, and the generation of action-specific events. Figure 2.9 shows a Workflow Engine that

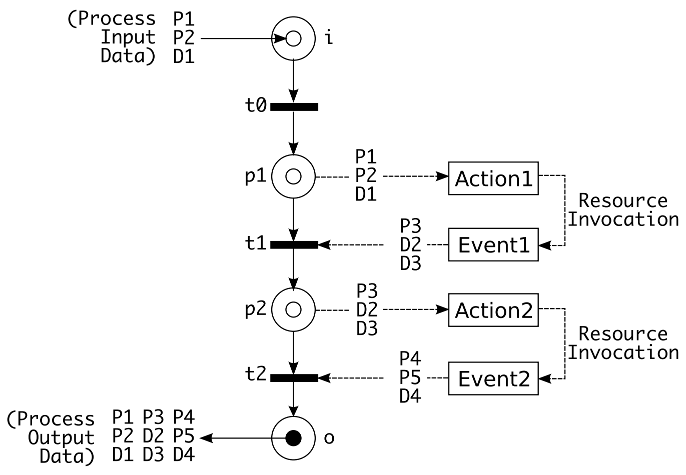


Figure 2.8 Actions and Events associated to places and transitions [FF04]

interacts with three external applications that expose the `INotifySink` interface, and receive events about a state change in the engine.

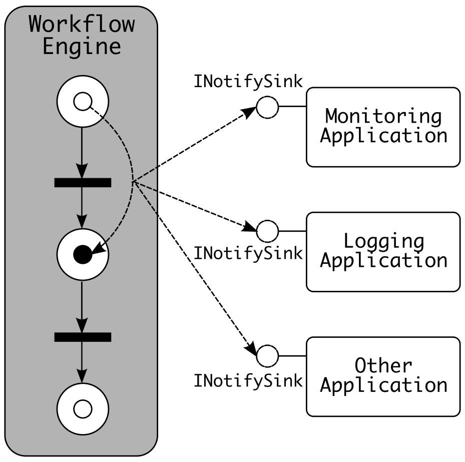


Figure 2.9 Extension to the functionalities of the workflow engine [FF04]

The proposal of Fernandes, Cachopo and Silva also uses a kind of intermediate language, but it has several differences with the aforementioned Workflow Kernel [FCS04]. In particular, the main motivation for their proposal is to cope with evolution and with the constant changes of workflow languages.

The core of their approach is a system that supports the execution of workflows defined with a *backend language*. This system is called the *Workflow Virtual Machine*, and it is complemented by a layer that transforms *frontend languages* into the backend language. The idea behind this separation is that frontend languages should be user-friendly and very suitable, and also that they may change frequently. On the other hand, the backend language is expected to remain largely unchanged and be able to support the execution of many frontend languages.

Besides proposing the general architecture for implementing this approach, Fernandes et al. also proposed a backend language. This language is a simple model that is capable of handling both control and data flow. To support concurrency, it also includes elements such as splits and joins.

These two proposals, the Workflow Kernel and the Workflow Virtual Machine, share the same problems: it is difficult to guarantee the expressive power of the intermediate language, and guarantee that it is going to support the high level / frontend languages. Furthermore, in the case of the Workflow Kernel, the intermediate language cannot be easily adapted to new requirements, although it is based on simple concepts that should be applicable to a large number of contexts.

Process Virtual Machine

The Process Virtual Machine (PVM) is an initiative of JBoss to build a reusable workflow kernel that can support various workflow languages, and which can be integrated into other applications [BV07].

There are four important characteristics behind the PVM design:

- **Flexibility:** the PVM should improve the collaboration between analyst and developers. This means that it should offer the flexibility necessary to implement very quickly all the changes to workflows required by analysts.
- **Extensibility:** the PVM should support multiple workflow definition languages, and it must be possible to extend the syntax and the semantics of such languages.
- **Embeddability:** the PVM is expected to be integrated into other applications. Therefore, it should offer the necessary mechanisms to integrate this system, control the workflows, and obtain information about their execution.
- **Plugability:** the PVM should allow the integration of configurable services that implement complementing functionalities. Some of these functionalities include persistence, transaction management, and testing facilities.

The PVM originated from the JBoss jBPM platform, and thus its execution model is almost the same model originally designed for jBPM. This model is similar to Petri nets in the usage of tokens, but it can also handle hierarchical tokens. However, this execution model lacks a formal specification of its semantics. Furthermore, the whole model has dependencies to Java and has some parts defined in terms of Java interfaces. Because of its origin, the PVM originally supported only jPDL (jBPM Process Definition Language). Moreover, support for BPEL was recently added, and more languages are expected to follow [OW208].

Mentor and Mentor-Lite

The Mentor project aimed at supporting an enterprise-wide workflow management platform involving both heterogeneous and distributed information systems [WWWKD96]. More in detail, its central goal was to develop a scalable and highly available environment for the execution and monitoring of workflows. They achieved this by proposing a solution based on an intermediate workflow language.

In Mentor, the formalisms to define workflow definitions include state and activity charts. These were selected because they are reportedly “perceived by practitioners as more intuitive and easier to learn than Petri nets yet they have an equally rigorous semantics" [WWWKD96]. On top of this formalism they were able to implement a system to verify workflow definitions using animation and simulations.

The Mentor executed environment is based on a distributed client-server architecture where a Corba-style object request broker is used to invoke application programs. In addition to the workflow engine itself, which executes the state and activity charts, the architecture of Mentor also includes a number of other components. These additional elements are necessary to provide additional workflow functionalities such as managing the list of tasks assigned to users, keeping log files with a registry of relevant events, and monitoring the execution of the processes.

Because of all these additional components, the Mentor architecture was regarded as heavy-weight and not suitable for every application. In response to this, the same working team developed a second version and called it Mentor-Lite. Mentor-Lite is a light-weight and tailorable architecture for executing workflows. This architecture is based on a small system kernel based on Mentor. However, this architecture supports extensions that implement additional workflow functionalities. The central difference is that in Mentor-lite these extensions are implemented as workflows. For instance, worklist management and history management have been implemented following this approach [WWWKD96, MWGW99].

2.4.2 Database based

The previously presented approaches employ reusable execution models that define both a way to represent run time state, and to progressively update that state. Other workflow systems have achieved similar results through the usage of database management systems. In these systems, the run time state of workflows is stored in databases, and the workflow engine updates that state according to the language semantics. The YAWL System's implementation is an example of this approach: it stores the state of cases in a database, and updates this state following the semantics defined for the YAWL language. Alternatively, some database based approaches rely entirely on mechanisms provided by the DBMS (e.g. triggers and stored procedures) to update the state of run time workflows, i.e. the engine is implemented *in* the DBMS.

The YAWL system that was previously mentioned can only be used to enact workflow definitions prepared using the YAWL language. We will now describe a database based workflow system that can be used with different high level modeling languages.

OPERA

OPERA is a basic kernel for supporting distributed workflow execution independently of the workflow language [AHST97b, AHST97a]. This kernel also shows that database technology can have a central role in supporting workflow functionality. Furthermore, OPERA focuses on some aspects of workflow execution that are not normally considered in detail, such as atomicity, persistency, and transactionality.

Two defining aspects of OPERA are the usage of database technology, and the usage of the OCR (OPERA Canonical Representation). These two aspects are discussed next.

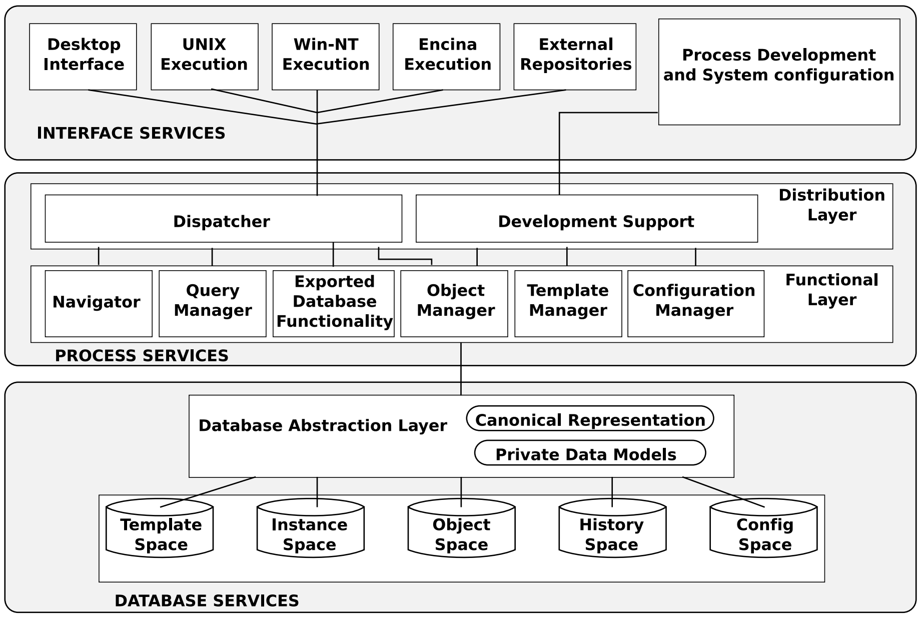


Figure 2.10 System architecture of OPERA [AHST97b]

The system architecture of OPERA can be divided in three parts as shown in figure 2.10. The first layer encompasses the *Interface Services* that allows the interaction between OPERA and other systems and services in several different platforms.

The second layer contains the components required for coordinating and monitoring the execution of processes. In particular, one of these components interacts with the data spaces of the third layer, one interacts with external objects, and one serves to query the status of the processes.

Finally, the third layer contains the actual repositories where data about processes is stored. This layer includes an internal abstraction layer that makes the rest of the system independent of the concrete database systems used. For example, this abstraction layer makes it possible to use an object store instead of a relational database.

The characteristics that set OPERA apart from other systems are mainly located in this third layer. For instance, the high availability and scalability sought by OPERA depends on the capabilities of the DBMSs used. Similarly, distributed workflow execution is achieved by leveraging the distribution characteristics of the DBMSs.

On the other hand, the responsibility of ensuring the transactional aspects of OPERA are shared between the process services and the database services. In the first place, OPERA relies on the transactional properties provided by the database systems used (i.e. atomicity, isolation, and durability). However, ensuring these properties also requires the active involvement of workflow specific services such as the *Navigator*, which acts as the overall scheduler, and enforces transactional aspects of the execution.

Figure 2.11 shows how OPERA was made independent of the workflow languages (application modeling languages). Instead of directly supporting each concrete language, OPERA defines and supports OCR (Opera Canonical Representation), a generic process model that plays the role of an intermediate model.

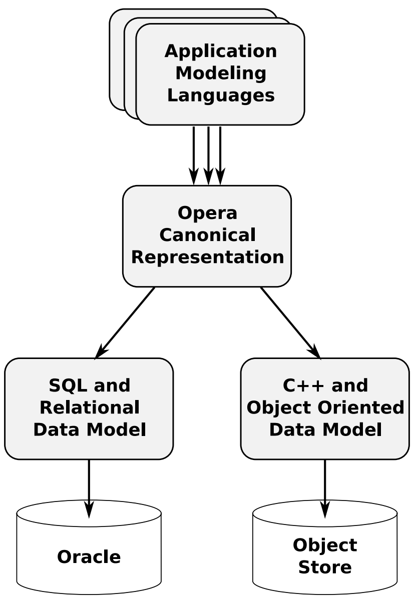

Application Modeling Languages
Opera Canonical Representation
SQL and Relational Data Model
C++ and Object Oriented Data Model
Oracle
Object Store

Figure 2.11 The different language representations in OPERA. Adapted from [AHST97b]

As such, OCR defines the concepts that can be managed (processes, activities, subprocesses, loops, forks, and others), their semantics, and a way of describing the models and storing their run time state. To be compatible with the rest of the approach, this canonical representation can be easily mapped to a relational database. On the other hand, the semantics of OCR models is similar to that of ECA rules (described in sections 2.2.2 and 2.4.4). This semantics is based on *guards* attached to tasks, *activation conditions*, and *input and output data structures* that describe and store the parameters and results of tasks.

2.4.3 Object oriented kernels

Another approach employed to support the development of workflow engines is the application of object oriented techniques. In these approaches, it becomes difficult to separate the implementation of the engine from the implementation of the language. Therefore, supporting several languages is more difficult than in approaches based on transformations to intermediate languages.

Among the approaches discussed in this section, the object oriented approach is the closest to the work presented in this dissertation. We will now present two very powerful object oriented kernels.

Micro-workflow

Micro-workflow is a kernel for software developers to introduce workflow functionality into object oriented applications [Man01]. Micro-workflow offers a lightweight architecture that enables these developers to select and adapt only the features that they need in their applications.

Micro-workflow is specifically intended to be used in object oriented applications. Because of this, it “applies techniques typical of object systems to solve workflow management problems. Consequently, it reduces the impedance mismatch between the provider of workflow functionality and application objects”

[Man01]. These techniques mainly refer to inheritance, extension, composition, and reusability.

The base of Micro-workflow is a set of workflow elements that implement basic control flow functionalities. In order to have domain specific behavior and structures, developers have to extend this base metamodel with their own specialized elements.

On the other hand, complementing workflow functionalities are implemented as components that interact with the control-flow core. Thus, developers can add new features by building more of those components. Also, they can tailor existing components if the features provided need to be adapted.

This approach provides two additional benefits. On the one hand, developers can choose only the features that they really need in their applications. On the other hand, these features are clearly isolated in components. Thus, they are easier to localize when they have to be maintained or removed.

Finally, we should mention two important limitations of the Micro-workflow architecture, with respect to other approaches discussed. The first one is that programming skills are necessary even to create the equivalent to workflow specifications. The second limitation is that Micro-workflow does not support high level standards such as BPMN or BPEL. Therefore, Micro-workflow is specifically targeted towards software developers.

Lewfe

Lewfe [Pér09] is an extensible workflow language and a lightweight engine, which can also be categorized as an object oriented kernel. Lewfe offers an open implementation of a workflow language, and enables the manipulation of the internals of the system. Furthermore, Lewfe is implemented on top of Cobro, a concept centric environment implemented in Smalltalk [Der06]. Because of this, Lewfe offers:

- A malleable implementation of the workflow languages, which can be modified at different points in time, including run time.
- An environment where domain knowledge appearing in the workflow language is reified in the application.

The metamodel of the Lewfe language is depicted in figure 2.12. This metamodel was adapted from [Man01] and it captures the elements necessary to manage sequences of activities, conditional and repeated paths, and also concurrency and synchronization of multiple control flows. Nevertheless, this language can be modified as necessary by modifying the metamodel or modifying the code of its elements.

Just as Micro-workflow, Lewfe also shows a merge between the implementation of the language and of the workflow engine. Because of this, most of the implementation of the engine is part of the implementation of the language. The Cumbia platform is similar to a certain degree, but the Cumbia Kernel and the Open Objects' framework on it offer several important functionalities that are reused in every element (e.g. coordination and synchronization). On the other hand, since Cumbia is implemented on top of Java, it lacks some of the dynamicity and flexibility provided by Cobro and Smalltalk.

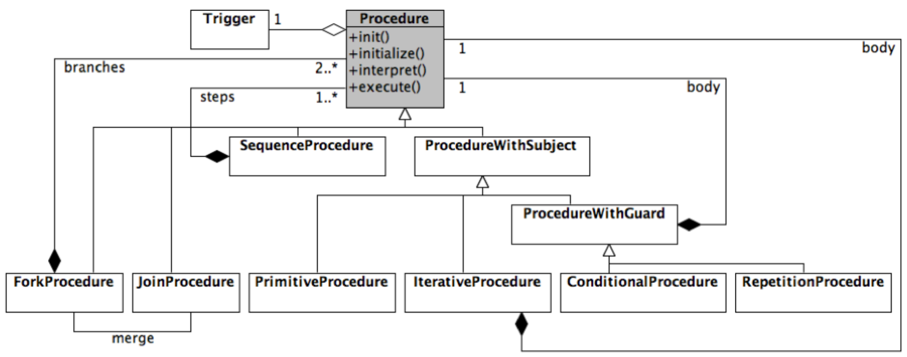


Figure 2.12 Workflow metamodel implemented in Lewfe [Pér09]

2.4.4 ECA Rules

To conclude this section we describe a system based on ECA rules called JOpera. As it was shown in section 2.2.2, ECA rules can be used to describe the semantics of a workflow language. In the case of JOpera, ECA rules are an intermediate step of the complete procedure to execute a workflow.

In reality, JOpera should not be considered a workflow kernel, because it only attempts to support one high-level or frontend language. Nevertheless, the approach that they use to execute the workflows has characteristics that make it worth presenting in this section. Furthermore, this approach could be adapted and used with other languages.

JOpera

JOpera is a visual composition language that provides a graphical notation to model workflows [Pau04, Pau09]. Originally, JOpera was intended to offer a visual extension to OPERA and to the OCR. However, the project evolved and it became an independent system that can use OCR as well as other execution environments such as BPEL or Java.

A workflow definition in JOpera is composed of three parts:

- A control flow graph that describes the control dependencies between the tasks of a process.
- A data flow graph that describes how data is transferred and transformed to be used in tasks.
- The description of the components that implement tasks' behavior, or the description of the services that are consumed.

In order to execute these specifications, JOpera performs two rounds of transformations. In the first one, the control and data flow are respectively converted into a set of ECA rules, and into a schedule of data transfers. This step also performs some verifications on the workflow definition, and can uncover inconsistencies.

In the second round of transformations, these elements are compiled into *executable* code. In his dissertation, Pautasso shows the mapping to convert JOpera specifications into OCR code, BPEL code, or Java code [Pau04].

Because of the double compilation step, the approach that JOpera follows is more complex than some of the other approaches discussed in this chapter. Moreover, JOpera also has two useful characteristics. The first one is that multiple executable models can be supported as long as a mapping can be found from the intermediate representation to the executable one. This characteristic can be used to select different execution environments depending on particular non-functional requirements. For instance, if transactionality is an important issue, OCR can be selected over BPEL.

On the other hand, the JOpera architecture can also be used to support other workflow definition languages. For this, it is necessary to find a mapping from the frontend languages into the intermediate representation. Since ECA rules have a high expressive power, many such languages are likely to be supported.

2.5 Development of engines for new languages

Based on the information presented in this chapter, we can now revisit the challenges identified in section 1.2.1. In the following chapter we will present the elements of the solution that we propose.

Limited extensibility and adaptability of engines and languages

The workflow specification languages presented in section 2.2.1 are a representative part of the spectrum of available languages. However, none of these languages includes powerful flexibility capabilities. BPMN and WS-BPEL offer a few extensibility points which are insufficient for complex requirements. One evidence of this is Sedna. It relies on WS-BPEL for its execution, but extends the language using different mechanisms than those offered in the specification. Those mechanisms are insufficient to support the extensions required in the language. Because of this, implementing Sedna required a larger effort compared to the effort required to implement a normal extension (e.g. BPEL4PEOPLE). With respect to this, YAWL and IMS-LD are also fairly limited, as the only flexibility points are the activities automatically executed or performed by users.

A further factor that limits the flexibility of these languages is the lack of modularization mechanisms. The tools presented in section 2.3 are more an exception than a rule, and they can be used only with a few languages. Nevertheless, it is important to note that some of those modularization approaches (MOBILE and AMFIBIA) explicitly state as a goal to support several notations. From the point of view of language flexibility, this is very important because it opens up the possibility of changing the languages as needed (at least their notations).

The limitations to extensibility and adaptability are not exclusive to the languages. Most of the engines and kernels that we have discussed also present important limitations. We believe that behind most of those limitations are the difficulties to modify the execution models. Among the kernels presented in this chapter, the more flexible ones are those where the execution model can be more freely modified, namely Micro-workflow, Lewfe, and, to a certain degree, the PVM. The key element for having this flexibility is allowing the usage of general purpose languages to define the execution models. On the other hand, there are kernels where the languages supported can vary but where the execution model is fixed (e.g. OPERA, Mentor, etc.). When those kernels are used, the biggest risk faced by language developers is that eventual changes to the languages may be impossible to support on the prescribed execution models.

Finally, it should be considered that most workflow engines are not developed on top of workflow kernels, but are instead built from scratch. Because of this, even the few flexibility capabilities offered by these kernels are not available to most engines.

Misalignment between high level model information and low level run time information

The workflow kernels presented in this chapter can be categorized in two broad groups. The first group contains kernels that use an intermediate workflow language or model. The second group contains the kernels where the modeling language used by workflow designers is directly executed.

The kernels of the first group are those where it is more likely to have a misalignment between the workflow definitions and the models that are effectively executed. On the one hand, this misalignment makes it difficult to map the elements of the high level models into elements of the execution model. On the other hand, this misalignment also makes it difficult to map the run time state of the execution model, into the high level model. A concrete consequence of this is an increase in the complexity of the tools associated to engines based on those kernels. This complexity is a consequence of considering those mappings. For example, a tool that monitors OPERA processes and shows the run time state to domain experts, has to be aware of at least the high level models, the OCR representations, and the mapping between them.

The second group of kernels, which is solely composed by Micro-workflow and Lewfe, does not have to manage that additional complexity. Their main drawback is offering a relatively low level approach. In the case of Lewfe, Cobro contributes to solving this issue. In the case of Micro-workflow, it is explicitly stated that high-level standards are not intended to be supported in the approach.

Limited support for developing new engines

In this chapter we have presented a number of workflow kernels that can be used to develop new workflow engines. Unfortunately, these kernels suffer from the two restrictions identified in section 1.2.1.

1. *Prescribed execution models*

   Most kernels support a single type of execution model that cannot be modified. The 'Workflow Kernel' of Ferreira and Pinto Ferreira is a good example of this, as all their approach is built on top of that particular execution model. As it was said above, the limited expressive power of any model creates the risk of encountering languages that cannot be supported.

2. *Fixed intermediate workflow languages*

   On the other hand, several of the kernels discussed support only one intermediate workflow language, and require high level languages to be converted into that language. These intermediate languages are usually low level, and thus have low suitability. Once again, the main issues of this are related to the expressive power of the languages, and to the additional complexity introduced into the tools. On top of that, there is the aforementioned problem of the lack of flexibility, which limits the capacity of those kernels to change and support languages that were not initially supported. Of the presented approaches, only the 'Workflow Virtual Machine' of Fernandes et al., and the PVM offer some kind of flexibility regarding the intermediate language.

These two restrictions of currently available workflow kernels are an important limitation to their applicability. However, these restrictions present their worst effects when languages evolve. Therefore, when initially selecting one of those kernels, a careful analysis of evolution requirements in the languages and in the domain has to be made.

Engines are coupled to languages

In this chapter we presented five different workflow languages that have different levels of tool support: for BPMN and WS-BPEL there are many workflow engines and tools available; for YAWL there is only one workflow engine; for Sedna there is no workflow engine, as it is transformed into WS-BPEL to be executed; and for IMS-LD there are a few engines that support subsets of the specification. In the majority of cases, the engines are specifically designed for those languages and they cannot be modified to support other languages. The only exceptions are engines built on top of a workflow kernel (e.g. a WS-BPEL engine built using the PVM).

In this respect, it must be said that engines designed for a specific language usually offer more powerful non functional requirements. One reason for this, may be that those engines are more mature, and are developed by better supported groups. Another possible reason is that those requirements are specific to the languages and are difficult to adapt to other languages. Nevertheless, some kernels also provide mechanisms to implement similar non functional requirements, e.g. OPERA and Micro-workflow. All this is also related to the following consideration about the tool chains associated to the engines.

Tool chains are not portable between languages

The final challenge identified in section 1.2.1 was related to the need of redeveloping entire tool chains for each new workflow language to be supported. In this chapter we have shown two aspects of this.

The first aspect is that workflow engines for existing languages already provide mature tool chains. However, these tools are usually tightly coupled to the languages and to the engines, and thus they are very difficult to adapt. The workflow reference model tried to reduce this problem by introducing standard interfaces, but they have not been widely adopted. In the case of WS-BPEL, the specification of the language also defines some interfaces that WS-BPEL engines must implement. However, these interfaces are usually complemented by vendor specific interfaces that offer more powerful services.

The second aspect shown is that workflow kernels can be leveraged to reuse a tool chain across a number of workflow languages. For instance, all the engines developed on top of the OPERA kernel can share the same tools, although they all rely on the OCR intermediate language.

3 Executable Models in a Workflow Kernel

The goal of this dissertation is to offer the means for developing workflow engines for extensible workflow languages with less effort than with state-of-the-art techniques. From a technical point of view, we propose a strategy based on using a *metamodeling platform* to define executable models, and using a *kernel* to execute multiple models in coordination. This strategy is based on a notion that we call *open objects*. From a methodological point of view, the strategy is to use concern specific workflow languages.

Cumbia is the concrete platform that implements this strategy. Cumbia's main constituents include a metamodeling platform to describe workflow language metamodels; a development framework to implement the elements of these metamodels; and an execution kernel to run executable models based on the open objects associated to these elements.

This chapter introduces the aforementioned constituents, and focuses on how each contributes to solving the problems previously identified. Subsequent chapters discuss those elements in more detail.

This chapter has the following structure. Section 3.1 shows how a metamodeling platform that uses executable models can be a base to implement workflow engines. Section 3.2 focuses on the execution kernel and discusses the role it has in the solution. In section 3.3, we discuss the idea of using concern specific workflow languages. Then, section 3.4 briefly presents the technique to coordinate the concern specific workflow languages, and shows what this implies in terms of the architecture of our solution. Section 3.5 introduces open objects and explains how they serve to implement the rest of the approach. Finally, section 3.6 introduces the Cumbia platform, which is a concrete implementation of the strategy presented in the previous sections.

3.1 A metamodeling platform for executable workflow models

The focus of this section are the main characteristics of our metamodeling platform, and the reasons to include them in our approach. This section starts by presenting some commonalities about metamodeling platforms and their application to the workflow context. Then, we describe the main properties of the metamodeling platform that we developed. The section is concluded with a discussion about the benefits provided by this specific platform.

Metamodeling platforms

A *metamodeling platform* is a modeling environment where the modeling language is specified with a metamodel that is open for adaptation. In a metamodeling platform, the modeling language, or modeling formalism, is not restricted to a standardized or predefined language. Instead, different languages can be defined, and each one can use the elements and constructs most adequate for the problems under consideration [KK02]. For example, in a metamodeling platform it is possible to build languages whose elements and structures closely reflect the structure of specific domains. There can even be a one-to-one mapping between elements of the domain and elements in the language. As a result, domain experts can use these languages very effectively, because they can relate their constructs to the domain knowledge that they posses.

In a metamodeling platform, metamodels are used to specify the abstract syntax and the semantics of the desired modeling languages. These are defined in a structured set of *metaclasses*, where each metaclass represents a modeling entity in the language and has an internal, encapsulated state [GKP98]. These metaclasses, and the relationships between them, define the *abstract syntax* of the language. On top of that, metaclasses also need to have semantics, which define the behavior of specifications created with the modeling language. Semantics can be described for each metaclass and be included in the metamodel definition, or it can be provided by external interpreters that run the models, or it can be defined in transformations. Finally, the concrete syntax of the modeling languages is also considered in a number of metamodeling platforms. This aspect falls outside the scope of this dissertation, and it is not addressed in our metamodeling platform. Instead, our platofm uses a common notation for all the languages that it supports.

A metamodeling platform for workflows

Our strategy to support the development of extensible workflow engines uses a metamodeling platform and relates metamodeling concepts to workflow concepts (see figure 3.1). On the top of the figure, there are metamodel definitions with are used to represent *workflow specification language* (e.g. BPMN or YAWL). Then, models conformant to those metamodels are used to represent *workflow definitions* (e.g. claim filling process, e-learning process for training '6042'). Finally, there are model instances, which correspond to *cases* in the workflow context (e.g. claim 1798 for client John Smith). Those instances are run time versions of the models.

Something to consider with respect to the platform is the process to select the elements that have to be part of a metamodel. We have to remember that we want to support workflow languages used by domain experts, and that we want to have domain concepts appearing at every stage of the life cycle of a workflow definition. Therefore, the elements selected to appear in a metamodel need to have a direct mapping to elements in the domain under consideration [KK02].

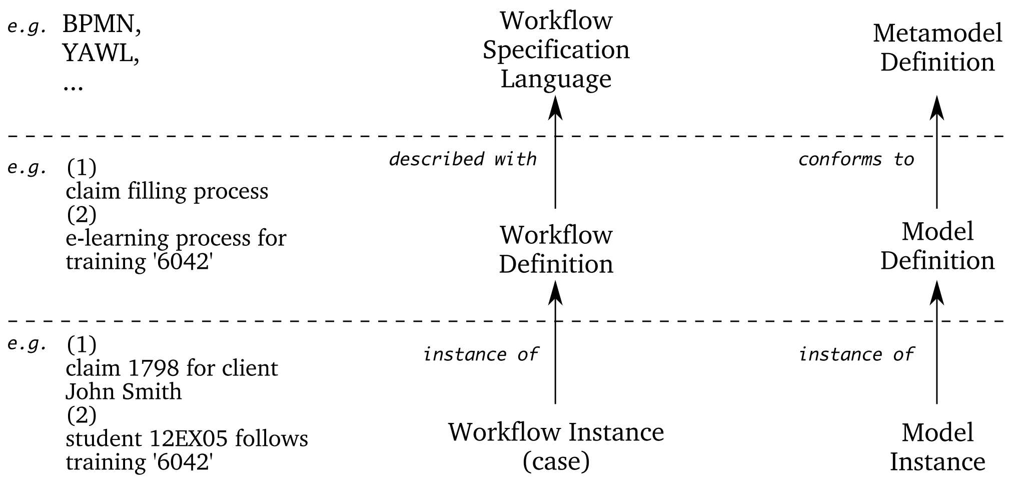


Figure 3.1 A metamodeling platform and the workflow context

To facilitate the following steps in the design of the metamodel, and to improve the extensibility and evolvability of both the metamodel and the language, the selected modeling entities (the language constructs) should have distinguishable semantics with respect to other entities [Cle07b]. This means that we will favor the identification of elements that have relevant behavior over elements that only have a structural role or only hold data.

A related aspect to consider are the means to define the semantics of workflow language elements. In some metamodeling platforms, the semantics of modeling languages are defined in a *translational* way [Cle07b]. This means that the semantics are defined by translations to different languages with known semantics. For instance, the precise semantics of BPMN depend on a mapping to BPEL that is described in its specification [Obj08]. Another example is UML, whose semantics is ambiguous until a specific translation to a programming language is defined.

As shown in chapter 2, existing workflow kernels rely on some underlying intermediate languages or execution models. Therefore, to be supported in a kernel, the workflow language needs to be aligned with the execution model of the kernel. This is a strong constraint that we wanted to avoid in our metamodeling platform.

Another consideration related to the specification of the semantics is the level of formality used. In many cases, semantics are only defined informally, using textual documentation. In other cases, more formal languages have been used to ensure that ambiguities and subjective interpretations are not possible. This is the case of Executable UML, which uses SMALL, TALL or the BridgePoint Action Language [MB02]. Other approaches have used formal mechanisms to define not only the semantics but also the structure of the modeling languages. For instance, Geisler et al. [GKP98] used Object-Z [DRS94] as a *metalanguage* to formally define the semantics of their metamodels. We later show that the semantics of our workflow modeling languages are specified for the most part with informal mechanisms. Nevertheless, the approach we utilize supports the majority of the concepts that a metalanguage has to support, according to [GKP98]: the concepts supported are Objects, Object Identity, Object Behavior, Classes, Compositionality, Inheritance, and Polymorphism; the only concept not supported are

Constraints. We analyze this further in section 3.5.

Our metamodeling platform is also characterized by making the definition of the semantics part of the metamodel definitions themselves. In particular, the definition of each metaclass in a metamodel includes the definition of its execution semantics (its *object behavior*). This definition includes two facets. The first one defines how instances of the metaclass should change their internal state in response to external stimuli, and how they should interact with elements external to the models. In the context of workflow specification languages, these external elements include, for example, other applications, data sources, or remote services. The second facet in the definition of a metaclass execution semantics determines how instances of the metaclass should interact with other elements in the same model instance (its *interaction behavior*). In our metamodeling platform we have kept these two facets as separated as possible. This is consistent with the *localization principle*, which dictates that each metaentity should have its own meaning as independent as possible form other metaentities [GKP98, Öve98]. Furthermore, keeping the internal object behavior oblivious of the interaction behavior contributes to lowering the coupling between language constructs, and to improving the flexibility of the metamodel [Cle07b].

A final distinguishing feature of the metamodeling platform is the explicit set of extension mechanisms provided [SJVD09]. These mechanisms can be used in two ways. Firstly, they can be used to define a new metaclass as an extension to an existing metaclass in the metamodel. Secondly, new metamodels can be defined as extensions to existing metamodels. In such cases, the new metamodels reuse the metaclasses included in the base ones, and specify new metaclasses, or extensions to the existing ones. In the rest of the chapter, and in the rest of the dissertation, we provide more information about how these features are supported.

Benefits of the metamodeling platform

Now we analyze the main benefits of using our metamodeling platform to support the development of workflow engines. The first benefit is related to having an almost one-to-one mapping between elements in the languages and elements in their implementations (the metamodels). Using the terminology of Cleenewerck [Cle07b], this means that the decomposition of language constructs is preserved in the language implementations. This contributes to improving the maintainability, extensibility, and evolvability of languages and their implementations, because they can co-evolve in sync and in terms of the same changing elements [Cle07b].

Another benefit of our metamodeling platform is to be able to define and support different workflow specification languages without the limitations found in other workflow kernels. Since the platform uses a generic execution model that is not tied to a particular workflow language or workflow execution model, the semantics of the languages supported in this platform are not limited.

From the point of view of the semantics of the languages, the platform also offers various benefits. In the first place, there are benefits related to the decision of making the semantics part of the metamodel definitions, instead of establishing it in additional components. For instance, in other metamodeling platforms the semantics of the languages are implemented with transformations that complement the metamodels. This is something useful in some contexts, such as MDA[Obj10], where a single model targets multiple platforms with different operational semantics. However, this strategy has drawbacks from the point of view of the flexibility and extensibility of the languages. This is due to the fact that to change the languages it is necessary to change both the transformations and the metamodels which may be structured in very different ways. On the contrary, if semantics are established directly in the metamodel definitions, then changes and extensions to the languages are easier to support. This makes it also possible to have a kernel that can be configured using only a metamodel definition. Also, since this kernel does not depend on any specific metamodel, extensions and changes to the metamodels do not affect its capacity to run the models.

In this section we have given an overview of the most prominent characteristics of our metamodeling platform and of the benefits that it intends to provide for the development of workflow engines. In the following sections we elaborate on these characteristics by showing how the models produced with the platform are executed in an execution kernel, and by showing how metamodels are implemented on top of open objects.

3.2 The role of a configurable kernel

The metamodeling platform that we just described can be used to specify the abstract syntax and the semantics of workflow languages. Using those metamodels it is possible to build executable models, which are run in a component called the *execution kernel*. The main responsibility of this component is to execute model definitions, but it also implements some related functionalities, such as managing persistence.

To enact a workflow definition (e.g. a claim filling process), an instance of the workflow is first created (e.g. claim 1798 for client John Smith). Then, the execution kernel successively updates the run time state of the instance, following the semantics of the language, until a final state is reached. In the meantime, additional tasks such as the following are performed: loading, transforming, and storing data; interacting with other applications; or reacting to external stimuli. To do all of this, the kernel assumes the basic functions of a workflow engine, which include *i*) creating and maintaining instances (cases) of the workflow definitions; *ii*) controlling the execution of the instances; *iii*) offering adequate interfaces to navigate workflow instances and their activities; *iv*) and offering interfaces to supervise and manage the instances [Coa99].

The previously described requirements can only be achieved if there is a kernel capable of interpreting the workflow definitions, which means that it has to be aware of the semantics of the language used to describe them. This is done by first instantiating the kernel and then configuring it with the metamodel that corresponds to the workflow specification language. This step makes it possible, in that particular instance of the kernel, to load workflow definitions, and to create and enact instances (cases). Note that a particular kernel instance can only be configured with one language at a time. As we will discuss later, supporting the execution of a workflow that involves multiple languages, is done by a kind of orchestration between different kernel instances.

In many workflow engines, the high level concepts appearing in a workflow specification languages are not available at run time. This is caused by the fact that in those engines high level workflow definitions are converted into low level representations, thus complicating activities such as monitoring or debugging. To solve these problems, traceability information that relates high level information with low level, run time information is frequently introduced. However, this also introduces additional complexity into the system because the workflow engine has to create this information and maintain it updated. On the other hand, the tools that depend on this information, such as monitoring applications, may have to correlate low level run time information with traceability information and with high level workflow definitions. Conversely, our configurable kernel always uses the elements defined in the metamodels. Therefore, no traceability information is necessary to relate the elements in a workflow definition to the elements present, at run time, in a model instances.

A second important point about the kernel are the additional workflow-related functionalities that it provides. These include persisting workflow definitions and the state of the run time models; doing basic conformance verifications; and mediating in the interaction with external applications. This is a subset of the functionalities commonly required in workflow engines, and the idea is that additional ones can be integrated as they become necessary. By offering them as part of the platform, less effort is required to support new languages. This is also an advantage with respect to current workflow kernels and engines because those functionalities are part of the kernel and are not entangled with the language implementations. It should be noted that the implementation of each of the functionalities included in the kernel is relatively simple; however, more complex implementations of these, and additional functionalities can be integrated at a later stage. As an example, one can consider persistence. The implementation currently included in the kernel only operates at the model level and can load and save the state of full run time models. If a finer granularity is required, for instance to persist sections of the run time models, the design of the kernel makes it possible to replace only the elements related to persistence functionalities, and put in place others that implement the finer granularity functionalities. This replacement is expected to be totally independent of the implementation of the languages' semantics.

Finally, the kernel has the additional responsibility of exposing interfaces to observe the execution and the state of the elements in a workflow instance. These interfaces are also necessary to allow the interaction of external applications with the instances run within the kernel.

The operations offered in the exposed interfaces can be classified in various groups. The first group comprises operations to navigate workflow instances and find particular elements. The next group comprises operations to query the state of elements at run time. Next, there are operations to change the structure of a model instance, or to change its elements, which in some contexts is a fundamental requirement. For instance, to support Level C of the IMS-LD specification [IMS03b] it must be possible to change the *learnflows* dynamically (see section 2.2.1). Finally, the last set of operations serve to control the execution of the elements of a model instance. All these operations are generic, i.e. they are not specific to a particular language, and they were selected to expose as much as possible of the run time elements and their run time state. Therefore, the platform can only make a few guarantees about the consistency of run time elements, and the rest of the responsibility is of the clients that use these operations.

Typically, each workflow engine offers a different set of operations which are strongly related to the supported workflow language. As a consequence, the applications that interact with an engine are strongly coupled to it and to the language it supports. This also means that those applications cannot be reused with other engines or other languages. On the contrary, all the operations provided by the kernel are independent of the metamodel used to configure it. Therefore, the exact same operations can be used with different workflow languages. The most important consequence of this is opening the possibility to reuse applications with different workflow languages.

In summary, the kernel is a component whose main function is to execute models built with metamodels that come from the metamodeling platform. This kernel is independent of the workflow languages, but it can be configured with metamodel definitions to understand the semantics of particular languages. Additionally, the kernel implements functionalities commonly found in workflow languages. Finally, the kernel offers powerful interfaces to interact with the models, which do not depend on the languages.

3.3 Concern specific workflow languages

In chapter 2 we showed the characteristics of some of the most common workflow languages. As we discussed, most of those languages do not use high level concepts that belong to a specific domain, but instead they use concepts that are general and belong to a technical level. Also, most of those languages model every aspect of a workflow within a single artifact. This has two negative consequences. Firstly, this artifact usually becomes complex and difficult to understand and to maintain; this happens even with relatively simple workflows. Secondly, the complexity of those languages is elevated, because they have to include a large number of elements, which are not always very related. Therefore, these languages can turn out to be difficult to implement, to learn and to fully use, especially when users are not technology experts but domain experts. For example, in the case of BPMN, most users never use more than a small fraction of its constructs (less than 20%) and never reach the level of detail that is necessary to enact or simulate the processes [zMR08].

In this dissertation we propose a solution to this problem that is based on two ideas. Firstly, there is the idea of modularizing workflow definitions according to concerns. Secondly, there is the idea of using, for each concern, a very suitable concern-specific workflow language. In the past, other approaches have modularized workflow definitions (see section 2.3). However, they used the same language for each concern. In this respect, the main difference of our proposal is the usage of different languages for each identified concern.

The first benefit of modularizing workflow definitions is the simplification of the artifacts that users have to manipulate. When monolithic languages are used, the definition of a workflow entails the construction of artifacts that mix many different elements. In the case of BPMN, these artifacts are diagrams that describe the control flow, the assignment of people to tasks, the management of exceptions, and the data consumed by tasks. Since all this information is combined, these diagrams can easily become big, complex, and difficult to manage.

Figure 3.2 shows the profusion of different elements that can be found in a relatively small BPMN process. This process only shows a subset of the elements available in BPMN to represent the control flow, the assignment of resources, time restrictions, and the flow of data. Version 1.0 of the BPMN specification

[Obj08] defined more than 50 constructs and attributes. The current proposed version 2.0 of this specification is even more complex and considers more kinds of processes [Obj09a]. Nevertheless, many of the constructs available in the language are rarely used in the practice [zMR08], thus raising the questions of how many of them are really necessary, and whether it would be useful to have a smaller language.

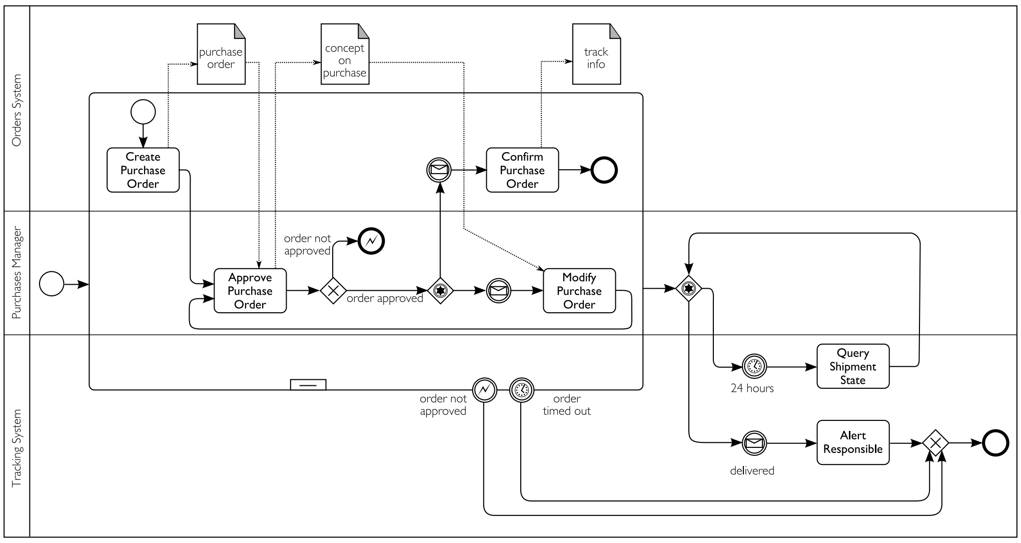


Figure 3.2 Sample BPMN process with control, resources, time, and data.

Modularizing workflow definitions leads to simpler artifacts that are much easier to create, to understand, to modify, and to maintain, although a cost has to be paid to assemble the parts back together. Modularizing workflow definitions also allows the usage of concern specific workflow languages – CSWfLs. These languages only target one particular concern of a workflow definition, and therefore they have a smaller set of elements that makes them very suitable for that concern. Concern specific workflow languages must be used in groups, in order to supply the complete semantics of a workflow definition.

An important characteristic of CSWfLs is to be simple, in the sense that they only handle small sets of elements. This is necessary since those languages are intended to solve problems restricted to a very specific concern. For instance, one can think of CSWfLs to describe the control flow in an e-learning application, or to describe how tasks are assigned to employees in a banking domain. In both cases, the number of concepts involved is considerably smaller than the number of elements in a language like BPMN. As a consequence of being simple, each CSWfL is easy to learn, and easy to implement and maintain [vDK98, vDKV00].

Another characteristic of CSWfLs is their suitability. Since these languages are designed to solve problems in very specific contexts, their elements can be high level and represent directly the important concepts of each concern. For instance, in the previously mentioned language to describe task assignment, users could be organized in an *organization chart*, instead of being organized in a generic tree or graph. Such a difference in terminology can be very important for domain experts. The suitability of those languages is also related to their simplicity: since they are simple to implement, it is more likely to have many of those languages implemented, instead of having only a few and forcing their use in situations where they are not fully suited.

Finally, CSWfLs can also be reused. Each of those languages embodies knowledge about a particular concern, which may be used in various applications [vDK98, vDKV00]. For instance, the language used to describe a billing concern could be used in a telecom workflow (like in [BVJ^{+}06]), or in a scientific application where grid services are payed per usage.

In summary, concern specific workflow languages are the strategy of our approach to make languages *simpler*, more *suitable*, and easier to *implement* and *maintain*. Nevertheless, it should be noted that the whole approach that we are proposing in this dissertation should work both with and without the usage of CSWfLs. In other words, the implementation of engines for monolithic workflow languages is also supported in our approach.

3.4 Coordination of concern specific models

The previous sections have focused on a metamodeling platform for executable models, a kernel to execute those models, and the usage of concern specific workflow languages. The combination of these ideas results in a metamodeling platform to define a metamodel for each concern specific language. The issue is then to *assemble* concern specific models and run them with the kernel as a whole entity.

We envision two possible strategies to do this. In the first one, the concern specific models are combined to make a *global* model that encompasses all the details about the workflow and are run in the kernel. However, this strategy has three problems. First of all, the execution kernel we propose is capable of running a model because it has access to the definition of the metamodel. In the case of a global model, either a *global metamodel* would be needed, or the kernel would have to deal with several metamodel definitions at the same time. The second problem is loosing the separation between models. Since users know about this separation, merging the models would be a way of loosing high level information. Finally, merging the models makes it more difficult to add or remove models late in the life-cycle of a workflow definition.

The second strategy that we envision does not use a global model and it is the one that we adopted and implemented. This strategy runs each concern specific model separately, in a different instance of the kernel, and *coordinates* their execution. By doing so, each kernel instance is configured with a single metamodel, metamodels do not have to be composed, and the separation between executable models can be maintained even at run time.

Figure 3.3 shows the main elements involved in implementing this strategy. In this figure we have used the term *assembly* to refer to a set of concern specific models together with information about how they have to be coordinated. This coordination information is specified with a language which is independent from the metamodels. To achieve this, the language has to describe the coordination in terms of basic elements provided by the metamodeling platform, instead of using elements defined in specific metamodels. Such a language makes it possible to have a single *weaver*, which interprets the coordination instructions and establishes links between model instances. The next section explains that these links are not based on the introduction of code, as weavers in other contexts do.

In our implementation, this language is called CCL, or Cumbia Composition

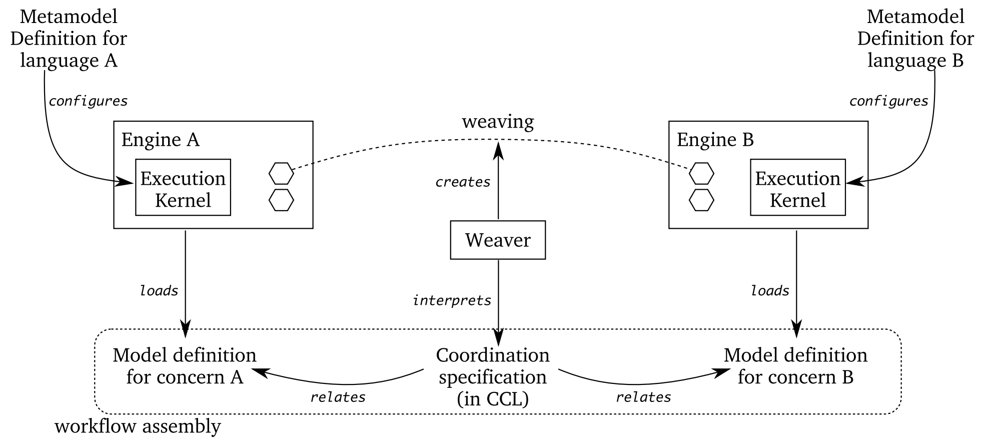

Metamodel Definition for language A
configures
Engine A
Execution Kernel
weaving
creates
Weaver
Metamodel Definition for language B
configures
Engine B
Execution Kernel
loads
interprets
loads
Model definition for concern A
relates
Coordination specification (in CCL)
relates
Model definition for concern B
workflow assembly

Figure 3.3 Coordination of concern specific models.

Language. As chapter 5 shows, CCL is a low level language that describes the coordination using only concepts provided by the metamodeling platform and the execution kernel. Because of this, CCL can be used to describe the coordination between any group of models. The only requirement is that they should be designed to run inside our kernel. Therefore, the definition of new metamodels does not require the definition and implementation of a new coordination language.

Finally, a further characteristic of this strategy, and of the weaver we implemented, is to support changes to the assemblies at run time. These changes include the addition and removal of models, and also the modification of the relationships. For example, an auditing model could be added to an already running assembly.

3.5 Open Objects

The goal of this section is to introduce the notion of open objects, and show how it is used as the base element to implement metamodels. This section also explains how open objects contribute to implementing the previously discussed approach. Chapters 4 and 5 provide all the other details about open objects.

Choosing the core elements of a metamodeling platform is critical because it determines the structure and semantics of the metamodels. For instance, in the case of UML, the concept of *class* can be considered the central element of the metamodel. In fact, most of the other concepts in this metamodel are directly or indirectly related to classes. The selection of the core elements determines the information required in the metamodel, and its structure. In the case of UML, using classes implies defining properties, operations, interfaces, and associations [Obj07a]. This is complemented by concepts to model the organization of classes, such as packages and components. From the behavioral point of view, UML models also describe interactions, messages, events, and state machines, which are also related to classes [Obj07b].

The role that classes have in UML, is the role that *open objects* have in our metamodeling platform. Classes and open objects share many common things, and open objects can even be considered as extensions to classes. The core of this “extension” is that the state that is normally encapsulated in an object, is externalized in a state machine. Figure 3.4 shows the basic components of an open object. The component marked as *entity* is a traditional object, with attributes and methods. It provides an attribute-based state to the open objects, and part of the open object's behavior is implemented in the methods of the entity. Each entity also declares a set of events that it can generate, and these events play a crucial role in the interaction of the open objects (see section 4.2.2) and the coordination of models (see chapter 5). Finally, the entity of each open object is not implemented from scratch. Instead, the Cumbia platform includes a development framework to implement new open objects on top of it. In this way, developers of new languages, and hence developers of new open objects and their entities, do not have to worry about things like generating events or handling the state machines.

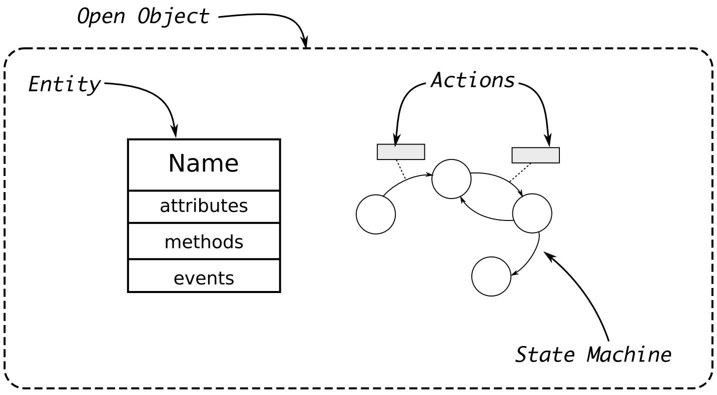


Figure 3.4 Basic components of an open object

Besides an entity, each open object also contains a *state machine* which represents one abstraction of its life-cycle. To materialize this state machine, developers have to identify the relevant states, establish the transitions, and make explicit how each transition is triggered. Furthermore, state machines also serve for coordination purposes. Whereas traditional objects can only interact through explicit method invocations specified in method bodies, open objects can observe other open objects' state machines and react to their changes. This interaction is based on the coordination of state machines and it is mostly external to the entities. Section 4.2.2 explains this in more detail. The main consequence of this is making the interaction more easy to analyze and modify.

The last element in an open object are *actions*, which are pieces of code related to transitions in the state machine. When a transition is taken, its actions are executed in a synchronous way. Actions offer the means to associate behavior to changes of state in an open object. Furthermore, actions do so in an explicit way, which is also easy to modify. This is even possible at run time.

Open objects offer essential characteristics to support the approach to build workflow engines that we present in this dissertation. These characteristics are the following:

- **Various types of execution models supported.** The behavior of each metamodel built in our platform, depends on the behavior of the open objects included in it. This behavior is determined by the methods in the entity, and the actions in the state machine, and it is not constrained in any way. Therefore, various types of execution models are supported.

For instance, using open objects it is possible to implement an execution model based on Petri nets: it is only necessary to implement the token management semantics in the entities, and establish the necessary interactions between the elements (see chapter 7).

- **Externalized state.** With the introduction of the state machines, the state of the open objects is externalized. This makes it possible for other elements to observe the state of the open objects, without resorting to using the methods of the entity.

- **Externalized interaction.** The state machines of the open objects offer the means to describe the interaction of an element in an externalized way. By analyzing these state machines, it is possible to know how each open object should react to changes on the open objects that surround it. Furthermore, this makes it much more simpler to modify the interactions of an element. Conversely, if the interactions were hidden in the methods of the entities, then they would be less explicit and harder to modify.

- **Compatibility with the kernel.** A kernel is capable of executing open object based models because the basic structure of the behavior is determined by the state machines structure. Therefore, the responsibility of the kernel is to coordinate the execution of the open objects of a model by managing the generation of events, distributing them, updating the state machines, and executing the actions.

  Another important point here is that the coordination mechanisms used by the kernel to synchronize open objects within a model, are also used to synchronize models corresponding to different workflow concerns.

- **A base development framework for developers.** To build an engine for a new workflow language, the developers have to define a metamodel and then develop its open objects. Since there is a framework where the basic behavior of open objects is already implemented, these developers can concentrate their efforts on the implementation of the semantics of their language. For instance, developers do not have to write the code to generate and distribute the events, to manage the state machine, or to execute the actions. Similarly, the kernel and the framework are responsible for handling the concurrent execution of several open objects, and remove this responsibility from the developer.

- **Explicit extension mechanisms.** Open objects define a series of extension mechanisms that make explicit the differences between an element and its extensions. These mechanisms, together with the fact that extensions to elements do not have an impact on the kernel, contribute to making the engines more extensible.

- **Decoupled interfaces.** The state machine, and the interaction mechanisms based on events, make it possible to interact with an open object without using an interface. This contributes to decoupling the elements, and facilitates the integration of external applications synchronized via event passing.

Compared to a metamodeling platform for UML or MOF, our metamodeling platform based on open objects is much more simple and includes less kinds of elements. Because of this, there are details about the open objects that cannot be included in the metamodel description and have to be introduced in the entity implementation. Nevertheless, the evaluation of our platform using the concepts defined by Geisler et al. [GKP98] shows that it includes most of the required concepts. According to the authors, a *metalanguage* should support the eight concepts listed below, and our metamodeling platform currently supports 7 of them. We have not yet addressed the last concept, *constraints*, but we believe that an existing constraint language, such as OCL [Obj06b], can be adapted in the near future. Note that although these concepts are defined in terms of objects and classes, they can be adapted to open objects and open object definitions. The following is a description of the concepts extracted from [GKP98], and a brief explanation about the way in which they are supported in our metamodeling platform.

- *Objects: encapsulation of an object's internal state.*

  At run time, each open object instance in a model instance has its own state. This state is encapsulated in the entities' attributes, but a view of it is offered through the externalized state machines. Nevertheless, the encapsulated state can only be changed by the entities' methods. Furthermore, the elements contained in different model instances are completely independent and do not have a shared state.

- *Object Identity: the notion of a persistent identity for an object.*

  Each open object instance in a model has its own identifier. At run time, it is possible to identify different instances of the same elements, which are contained in different instances of the same model. For example, if the process of figure 3.2 is instantiated three times, there are three instances of the activity "Create Purchase Order", each with a different identifier.

- *Object Behavior: the possibility of objects to evolve in a pre-defined way.*

  Each open object defines its own behavior, which is implemented in the methods of its entity and its actions. This means that each open objects defines how its state should evolve in response to external stimuli such as event notifications and method invocations. The 'pre-defined way' mentioned in [GKP98] points out that the behavior of elements is intrinsic to them, and it is not something provided by the context.

- *Classes: the ability to describe the common aspects of objects and encapsulating them in a class structure.*

  In our metamodeling platform, an open object definition is the equivalent of a class in UML: it defines the structure and behavior of a multitude of open objects instances used in models. In other words, open object definitions work as classifiers [Obj07a].

- *Compositionality (associations): the ability to declare a structural connection between classes [...].*

  The definition of a metamodel includes the definition of associations between open object definitions. These associations can have simple or multiple cardinality. Furthermore, in associations with multiple cardinality the links can be ordered or they can be named. This characteristic is very important to allow model navigation.

- *Inheritance: defining a class as an extension of one or more existing classes.*

  Our metamodeling platform offers the possibility of defining new open objects as extensions to existing ones (see section 4.2.5). For instance, in a BPMN metamodel there can be a generic open object called Task, and also extensions to it to represent Manual Tasks or Automatic Tasks. In addition, our metamodeling platform also allows the specification of *abstract* open objects which must be extended to be instantiated.

- *Polymorphism: the ability to define operations that act upon several distinct classes.*

  In our platform, polymorphism is related to inheritance. We do not have a concept similar to that of interfaces in Java, but we can have *abstract* open objects that *must* be specialized to be used.

- *Constraints: the ability to describe constraints between objects or [...] objects operations [...].*

  The only concept defined by Geisler et al. that our metamodeling platform does not currently support are constraints. In UML, this concept was implemented with the introduction of OCL, and to support it in our metamodeling platform would require a similar effort. We have envisioned two alternatives to achieve this. The first one is to develop, from scratch, a new language to describe constraints that are suited for our specific platform. The second alternative is to start from an existing language, such as OCL, and adapt it to the include the aspects that are specific to the open objects. This alternative will probably require the complete implementation of a new OCL checker, as we do not expect to be able to adapt one of the existing ones to our platform.

3.6 The Cumbia platform

The previous five sections presented our strategy to support the development of workflow engines for new workflow languages. We implemented this strategy in a concrete platform that we called Cumbia. More specifically, this platform was implemented in Java, supports Java-based open objects, and uses XML as the representation for most descriptors, including metamodel and model definitions. This platform is composed by a number of elements that we now introduce (see figure 3.5). Chapters 4, 5 and 6 present all of those in more detail.

The first element of the Cumbia platform is a concrete implementation of the *metamodeling platform*. This implementation offers various elements to define and use the metamodel and model definitions. In the first place, the platform offers the means to load these definitions and to perform some consistency checks. For instance, it can verify whether a model is conformant to the metamodel it claims. Consequently, the platform also establishes a format that metamodel and model definitions have to follow. To facilitate the interaction with other tools, these formats were defined using XSD schemas (see appendix B).

In a similar fashion, the metamodeling platform also offers the means to describe and implement open objects. We refer to this as the *open objects framework*

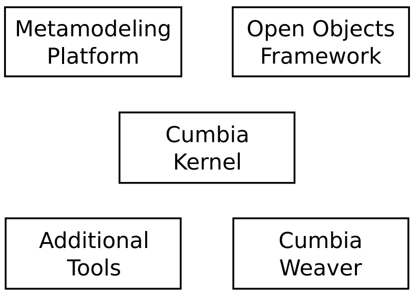


Figure 3.5 Main elements of the Cumbia Platform

and it can be seen in two parts. On the one hand, the metamodeling platform defines the format of another XML-based descriptor which is used to specify some aspects of the open objects, such as the structure of the state machines. On the other hand, the metamodeling platform also includes a development framework that serves to implement the behavior of the open objects.

Another fundamental element in the Cumbia platform is a configurable execution kernel that is compatible with the described metamodeling platform. This component is called *Cumbia Kernel*, and it is a concrete kernel to execute models based on open objects. To achieve this, the Cumbia Kernel has to be first configured with a metamodel definition. This definition should also include the structure of the relevant open objects, and the implementation of their behavior. After this configuration step, the Kernel is then capable of executing the models by controlling the execution of the open objects. This is mainly achieved through the generation and distribution of events (see section 4.4 for more details). Finally, the Cumbia Kernel also implements basic workflow functionalities, and offers interfaces to interact in many ways with the model instances at run time.

The next component in the Cumbia platform is the *Cumbia Weaver*. The main responsibility of this component is to interpret coordination programs written with CCL. By doing so, it establishes links between elements located in different model instances. This is possible for the Cumbia Weaver because the Cumbia Kernel offers the interfaces that are required to build those links. Section 5.3 provides more details about the responsibilities of the the Cumbia Weaver.

Finally, the Cumbia platform also includes a number of *additional tools*, which are not essential to use the approach. However, these tools make the approach more usable to language developers, and improve the quality of the results. These additional tools, include an editor for metamodels and for open objects, a framework to develop tests for new workflow languages, and a debugger to find problems with the models at run time. Chapter 6 describes these tools in more detail.

3.7 Summary

This chapter introduced the core elements of the proposal, which include a metamodeling platform to describe workflow specification languages, a kernel to execute models conformant to those metamodels, and a modeling abstraction to describe the elements in the metamodels. This dissertation also proposes the modularization of the workflow specification languages with the introduction of concern specific workflow languages, and it provides the means to integrate concern specific models by describing how their execution should be coordinated.

This description of the coordination is written with a language called CCL and it is external to the models and to the metamodels.

The content presented in this chapter is discussed in more detail in the chapters that follow. Chapters 4 and 5 focus on describing all the elements that are part of the proposal and include all the relevant technical details. Chapter 4, describes the characteristics of the open objects, their mechanisms of coordination, and how they can be extended. Then it shows how they are used in the metamodeling platform to form metamodels representing workflow specification languages, how engines are structured around the Cumbia Kernel, and how these engines use the metamodel specifications to enact workflows. Chapter 5 complements this by focusing on the usage of multiple concern specific workflow languages. This chapter describes how the coordination of concern specific models conforming to different metamodels is achieved by means of CCL and of the Cumbia Weaver. Having discussed all the technical details of the proposal, 6 presents the structure of a development process to develop workflow engines. Chapter 7 shows how the ideas presented in this chapter were validated in a number of case studies.

4
Workflow Models based on Open Objects

In the previous chapter we introduced the main elements of the approach, and we justified their usage in relation to the problems identified in chapter 1. The goal of this chapter and of chapter 5 is to concretize what was introduced in chapter 3, by providing all the details about the design and usage of the Cumbia platform and of the proposed approach. We have divided the presentation of this between this chapter and chapter 5. First, chapter 4 presents the notion of open objects and shows how to use open objects to define metamodels. This chapter also presents, in detail, the mechanisms that make the models executable, and illustrates the most important concepts with a workflow language called `MiniBPMN`. Chapter 5 continues the presentation by explaining and illustrating how multiple concern specific workflow languages can be constructed and have their executions coordinated.

Chapter 4 is organized as follows. Section 4.1 presents the general features that metamodeling platforms should offer and their relation with executable models. Section 4.1 concludes by introducing open objects as an element to implement these features. Then, section 4.2 presents the Open Objects in detail. This includes an in-depth presentation of the components of open objects, their interaction mechanisms, and of the concrete means to specify and implement new open objects. Using this information, section 4.3 shows how metamodels are built using open objects, and how model definitions are written. Section 4.4, focuses on the execution of models. For this, it first discusses the way in which models are instantiated, and then it explains how the Cumbia Kernel executes those instances. Section 4.5 explains how an engine is built on top on Cumbia, and discusses the architecture of Cumbia based workflow engines.

The rest of the sections (4.6, 4.7, and 4.8) are devoted to illustrate the ideas previously exposed. These sections use a workflow language called `MiniBPMN` and show how a metamodel for this language is specified and implemented, how models conformant to the metamodel are defined, and how they are executed. To close the chapter, section 4.8 shows how `MiniBPMN` can be extended with new concepts, and then shows how these concepts are introduced as extensions to the base metamodel, and the small impact that these extensions have on it.

4.1 A metamodeling platform based on Open Objects

4.1.1 Metamodeling platforms

The general goal behind model driven development is to "use modeling techniques to bridge the gap between the problem domain and the software implementation domain" [FR07]. Different strategies and technologies have been applied to address this goal, but most of them have a common characteristic: to use fixed metamodels (OMG level m2). This means that, under these approaches, all models are conformant to the same metamodels, or are created using the same *language* and base elements. Therefore, the flexibility in those approaches has been limited to the model level (OMG level m1) [KK02].

To counter this, metamodeling platforms have appeared. In those, flexibility is introduced at an additional level of the typical metamodeling pyramid. Instead of having a fixed metamodel (m2) to produce all the models (m1), the metamodel can also vary. By doing so, the overall platform is more flexible and it becomes possible to further narrow the gap between the problem domain and the solution domain. The fundamental idea behind all of this is that flexibility at the metamodel level reflects in models that are expressed using concepts that are closer to the problem domain and thus create more suitable solutions to the problem. Some examples of metamodeling platforms include the Eclipse EMF [The10a], the MIC/GME [LBM$^+$01], Memo [Fra02], and several other MOF-based platforms.

The design of a metamodeling platform has to take four aspects into account. First of all there are the means to describe the metamodels. Then, there are the means to describe models conformant to the metamodels previously created. The third aspect includes the operations that can be performed on the models. Finally, there are the technical and practical details of the platform, such as the technologies involved, the tools supported, and the generic architecture of applications based on the platform. We now describe this four aspects, and in later sections we will provide concrete examples of these.

A metamodel description, which is usually created using a meta-metamodel (OMG level m3), defines a collection of *types of elements* to use in models, or metaclasses. The description of each type should include its name, the attributes of its instances, and the relations allowed with other type instances. In some platforms, constraints can also be specified (e.g. using OCL in the case of UML). This creates the restrictions that are necessary to later check if a model is conformant to a given metamodel, or not. In section 4.1.2 we will discuss about associating behavior to these types.

An issue faced when designing a metamodeling platform is to provide adequate modeling abstractions to describe the metaclasses. The number of metamodeling platforms based on the MOF meta-metamodel (OMG level m3) [Obj06a] shows that the modeling abstractions do not necessarily have to be complex in order to be useful. However, if more advanced information about the types had to be specified, then MOF would be dropped and more expressive modeling abstractions would have to be developed.

After the artifacts to specify the metamodels are selected, the following issue is to put the metamodels to use. To achieve this, the metamodeling platforms have to offer the means to describe particular problems of the domain using models conformant to the metamodels. In general, a model specification describes instances of types and relates them following the restrictions established in the metamodel.

Models themselves are not very useful if the platform does not offer operations to query or interact with them. These operations can have very different natures, and they can be applied to one or several models at a time. One example of operations are those that offer access to the structure and information in the models. Another example are operations that transform models or that generate code or documentation taking as input one or several model specifications (e.g. MDA [Obj10]). Other operations serve to weave models (e.g. AMW [FBJ$^+$05]). Finally, some operations are useful to validate that models are conformant to the metamodels.

Finally, the previous three aspects are complemented in a metamodeling platform by an underlying infrastructure and tools that metamodels and model developers can use. This results in considerable savings in development costs as the tools and architectures are easily reused [KK02].

4.1.2 Model behavior

In this dissertation we are not concerned with models that represent static and stateless structures. Instead, we are interested in stateful executable models: a model is executable if it is possible to compute how its state changes in response to input data [HBMN07]. This creates further requirements for the metamodeling platforms as they have to offer the means to describe, at the metamodel level, the behavior expected to be seen in models.

There are at least three strategies to define this behavior. The first one is using *translational semantics* [Cle07b]. This means that the behavior is indirectly defined by establishing a translation schema to a metamodel with well known semantics. For example, the specification of BPMN includes a chapter that fixes the behavior of BPMN processes by providing a mapping to WS-BPEL processes. Another example are languages that rely on code generation to be executable. Although this is an effective approach, it also has problems because the behavior definition ends up scattered between the transformation rules and the target language.

The behavior of models can also be specified by directly defining how their state should be updated in response to stimuli. For example, graph rewrite rules [BFG95] can be used to specify model changes (including state changes or changes to their structure) at different levels, from single elements to entire models.

Finally, the behavior of models can also be specified in an object oriented fashion by encapsulating in each element its own behavior definition. This definition should include the following three aspects: *i)* the way in which internal stimuli force changes to the state of the element; *ii)* the way in which the element should create new stimuli for other elements of the model in response to stimuli received (the interaction behavior); *iii)* the way in which the element should interact with the execution environment (e.g. the elements and applications external to the model).

Each metamodeling platform that aims to support executable models has to provide a strategy to define the behavior of the models. Since each one has advantages and disadvantages, the strategy should be selected according to the particular domains and the kinds of applications constructed. The rest of the chapter shows that the Cumbia platform employs a strategy that is close to the object-oriented one that we just discussed.

4.1.3 The role of open objects

Open objects are the artifact used in Cumbia to describe the elements of the metamodels. On the one hand, open objects serve to specify the element types and the structure of the metamodels. On the other hand, they serve to describe the behavior of those elements in order to make the models executable.

Each element type in a Cumbia metamodel is represented with an open object. Open objects can have attributes, which serve both to configure the element and to maintain its state at run time. Open objects can also be related using inheritance and aggregation relations. As a result, new element types can be created by extending existing ones; similarly, aggregations restrict the structure of the models that want to be conformant to the metamodel. However, it should be noted that the inheritance and aggregation relations between open objects have a special semantics that is studied in the next section.

Finally, each open object can specify its own behavior, its interactions with other open objects, and the mechanisms to interact with the execution environment. An engine can then take this specification and use it to successively update the state of a model instance in a way that reflects the semantics of the language.

The following sections will provide a complete description of the open objects, of their interaction mechanisms, and of their characteristics to create extended element types. This is illustrated with a new metamodel called `MiniBPMN`.

4.2 Open objects

The goal of this section is to present the notion of Open Objects and discuss their most important characteristics. This section is divided in five parts that present different aspects about open objects. In section 4.2.1, the general characteristics and the main elements of an open object are introduced. Then, section 4.2.2 part presents in detail the interaction mechanisms used to synchronize the execution of open objects in a model. A navigation language to describe and find elements in open objects' graphs is presented next in section 4.2.3. Afterwards, section 4.2.4 shows concretely how an open object is described and implemented. Finally, section 4.2.5 presents and illustrates the mechanisms that allow the definition of new open objects as extensions to existing ones.

4.2.1 General characteristics

Open objects are the abstraction used in Cumbia to model the elements of a workflow language. By using this abstraction, it is possible to represent the structure, the state and the behavior of those elements. Furthermore, this notion offers various mechanisms to model the interaction between those elements, for instance as a reaction to state changes. Figure 4.1 shows the basic components of an open object.

Entity

The central element of each open object is what we have called the *entity* of the open object. Basically, an entity is an object (a standard object, i.e. a Java object) which has three main purposes. First of all, most of the behavior of the open object is implemented in the methods of the entity. For this, each open

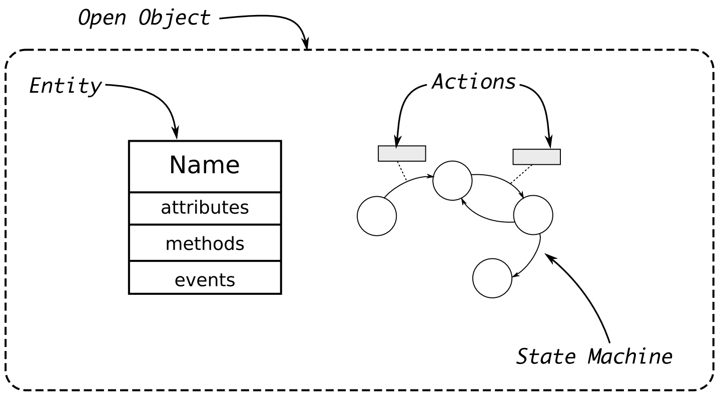


Figure 4.1 Basic notation to represent open objects

object declares the set of methods that it supports, using an interface that is implemented by the entity. The second purpose of the entity is to maintain part of the run time state of the open object. This is done with the attributes of the entity, and are complemented by the state machine.

Finally, the entity generates events that are required to synchronize various open objects, or to synchronize the entity with its state machine. The role played by those events is analyzed in section 4.2.2.

As an example, consider an open object called `Task`[1], whose entity is shown in figure 4.2. This open object is used in a workflow language where the execution of tasks follows rules similar to those found in a Petri net: at a given time, only the tasks that posses a token can be executed. Because of this, the entity of `Task` has a boolean attribute called `hasToken`, which indicates if the task currently posses a token and thus can be executed. The entity also implements four methods that are relevant for the semantics of a `Task` in the context of this workflow language: method `setToken()` makes true the attribute `hasToken`, while `removeToken()` makes it false; method `activate()` starts the execution of the concrete task that has to be performed (e.g. consuming a remote service, processing data, requesting inputs); and `finish()` announces that the concrete task has been completed. Finally, this entity also declares that it can generate two events, namely `tokenReceived` and `finished`. Listing 4.1 shows the general structure of the implementation of this entity. The methods of the entity are declared in the interface `ITask`.

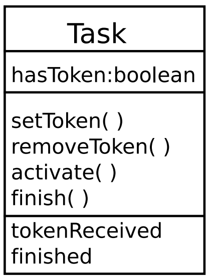


Figure 4.2 Entity for the open object `Task`

[1]In this section we will use `Task` to illustrate, element-by-element, all the constituents of an open object. In section 4.2.2 we will show how `Task` is related to other elements in a metamodel.

Listing 4.1: The implementation of the entity of `Task`

```
public class Task extends OpenObject implements ITask
{
  private boolean hasToken;

  public void setToken( ){ ... }

  public void removeToken( ) { ... }

  public void activate( ) { ... }

  public void finish( ) { ... }
}
```

In ensuing sections we will show how entities are described, configured, and implemented. In particular, in section 4.2.2 we will show how and when the events declared in an entity are generated. Section 4.2.4 shows how each entity is actually an extension of various existing classes provided by the Cumbia framework. By providing these base functionalities, the amount of code required to implement a new entity is greatly reduced and focuses on the domain and on the requirements of the particular language.

State Machine

Each open object also contains a *state machine* which materializes one of the possible abstractions of its life-cycle. Compared to the states identified by the attributes of an entity, the states in a state machine are coarser and thus can be in a higher level of abstraction. Furthermore, since those states of the open object are reified in the state machine, it is easier to react to state changes.

The states in a state machine are connected by transitions and each transition has an associated trigger event. When the open object receives an event that triggers a transition, the open object changes its current state and may trigger the execution of some code (see *actions* below). This is the basic mechanism used to execute a model, and it is fully described in section 4.4.

Figure 4.3 shows the state machine for the open object `Task`. This state machine has two states, *Waiting* and *Active*, which are connected by two transitions. Transition *activation* is triggered by the event `tokenReceived`, and it has an associated action called *activate*. Transition *deactivation* is triggered by the event `finished`, and it has two actions associated, which are called *removeToken* and *giveToken*. In section 4.2.2 we will explain that the string `[ME]` indicates who is expected to generate the event.

The structure of a state machine is specified with an XML file used in the context of a metamodel. The structure of such file is explained in detail in section 4.2.4.

Actions

Actions are typically small pieces of code associated to transitions of a state machine. Using actions it is possible to associate behavior to the state changes: when a transition is taken, the actions associated to it are executed. By default, actions are executed synchronously, one after the other, following the order

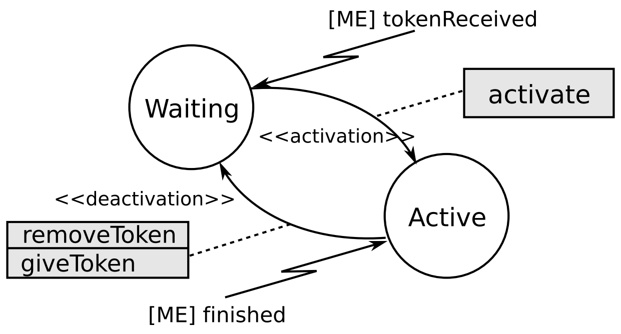


Figure 4.3 State machine for the open object `Task`

specified by the designer of the metamodel. However, it is also possible to mark actions as asynchronous and then execute them in parallel to the other ones. This is explained in detail in section 4.4.

The state machine of the open object `Task` (figure 4.3) includes three actions that are executed when state changes take place. The action *activate* invokes the method `activate()`[2] of the open object when the transition *activation* is triggered. The actions *removeToken* and *giveToken* are executed in that order when transition *deactivation* is triggered: action *removeToken* invokes the method `removeToken()` of the open object, and action *giveToken* invokes the method `setToken()` in the open object that follows the `Task`.

Memory

A final aspect about the open objects is something that we have called the *memory* of the open object. Without this memory, the only way to store information in an open object is to use the attributes of the entity. However, this may be problematic because the only way to access this information is to use the methods offered by the entity. Furthermore, including additional fields or modifying the existing ones requires altering the implementation of the entity.

To solve these problems, every open object has a memory, which is a dictionary based data container. This memory can be accessed using a simple interface that is common to every open object. All the information stored in the memory of an element is named, and it must be an instance of either some basic or some composite data types. The basic types supported are String, Integer, Double, Boolean, and Byte. The composite data types are Properties or Sequences. In retrospective, these data types, including their representation as strings, are very similar to JSON [Cro06, jso10].

The memory of the open objects is used for two reasons. The first one is to configure the elements in a model by providing some initial values (see section 4.3.2). The other is to temporally store information about the model execution, such as partial results.

For example, consider a workflow definition where different instances of `Task` are used to consume different web services. In order to configure the locations of these services, the workflow definition indicates the values to include in the memory of each `Task`. Now consider a monitoring tools that queries each `task` to

[2]It is only by chance that the name of the action and the name of the method coincide, i.e. the method invoked is not inferred but it is explicitly indicated in the code of the action.

know which service it is consuming, and how many timeouts it has encountered during the execution of a case. From the point of view of the implementation of the kernel and of the monitoring tools, it is easier to store this information in the memory and to offer a standard interface to make the queries, than using a reflection API to query the attributes of the entity.

Relations

Finally, there is another detail about the open objects worth mentioning briefly at this point. Similar to UML's associations, it is also possible to establish *relations* between open objects. Relations play a very important role in the construction and execution of the models, and they have several important characteristics. In the first place, relations are named and typed. Relations also have a cardinality, which can be 'simple' or 'multiple' (they are equivalent to UML's '0..1' and '0..*', respectively). Furthermore, relations where the cardinality is 'multiple', can be implemented in two ways: as 'sequences' or as 'maps'. The conformance of a model to a metamodel can be validating by considering the relations defined in the metamodel and the way in which they are instantiated in the models.

Sections 4.2.4 and 4.3, will present more information about the meaning and the definition of relations. Additionally, the navigation language that will be presented in section 4.2.3 has lots of dependencies on the relations.

4.2.2 Open objects interaction

There are two mechanisms of interaction between open objects. The first one is synchronous and uses direct method calls to invoke methods in the entities. These invocations can be found in three locations: *i*) in the body of actions; *ii*) inside methods of the entities; *iii*) or they can come from elements external to the metamodels.

The second mechanism of interaction is based on events that trigger transitions in the state machines. This is possible because each transition has an associated trigger event, which is specified with a source and an event type. When an event matching the description is received by the open object, the transition is triggered. Then, the actions associated to it are executed, and the state machine changes its current state. Events are not processed immediately, but are arranged in a queue and processed asynchronously. Section 4.4 explains in detail the mechanisms to process these events, including the way in which the queues are managed.

There are two kinds of sources for these events. The first source are the methods of the entities, which can explicitly generate specific events. To support this, the open objects framework offers methods that can be invoked from within methods to explicitly generate events. Nevertheless, an entity can only generate events declared in the description of the open object (see section 4.2.4). Listing 4.2 shows a method in an entity which explicitly generates an event.

The second source of events are the state machines themselves. When a transition is triggered in a state machine, several events are generated to keep other elements informed. Figure 4.4 enumerates those events, which signal the different steps in the processing of a transition.

1. `Exit State`: the first event generated announces that the state machine is

Listing 4.2: Generation of an event in an entity

```
public void activate( )
{
    ...
    generateEvent( "activated" );
    ...
}
```

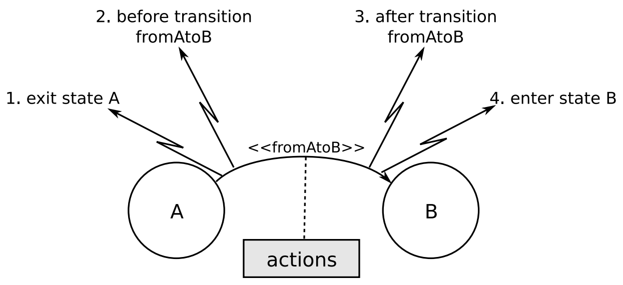


Figure 4.4 Events generated when a transition is triggered

leaving a certain state. If several transitions depart from the same state, the first event generated is the same.

2. `Before Transition`: the second event generated announces that a specific transition was triggered. This event is generated before the actions associated to the transition are executed.

3. `After Transition`: the third event generated indicates the completion of the execution of the actions associated to a specific transition.

4. `Enter State`: the fourth and last event generated announces that the state machine reached a certain state. If several transitions arrive to the same state, the last event generated is the same.

In simple cases, some of these events may be redundant. For instance, when a state has only one outgoing transition, the first and second events are equivalent. Similarly, when the transition has no actions associated, the second and third events are also equivalent. Nevertheless, in more complicated situations the four events are necessary to have the adequate fine grained synchronization hooks. Furthermore, it is important to note that these events are generated and delivered in the order depicted in the figure. Therefore, it is not possible for a listener to receive them in a different order.

An important detail about trigger events associated to transitions is the way they are described. Each trigger event is described with an expression of the form `[source] eventName`. In this expression, `eventName` serves as an identifier for the type of event. For instance, in figure 4.7 the type of the events expected are `tokenReceived` and `finished`. In this type name there is no indication about the origin of the events: they could be generated by an entity, or they could be generated by a state machine.

On the other hand, `[source]` is an indicator of the expected source of the event. Therefore, events are not filtered only by type, but also by the open objects that generate them. This works as a kind of *message filter* [JVR97, AWB$^+$93], and it is the inverse of *acquaintances* in the actors model [Agh86]. This means that instead of specifying who can receive a given event generated by an open object, we specify the open objects that can send that particular event to it.

To specify these filters we use *roles*. In this context, a role is an abstract description of *sets* of elements in a model, where each set is relative to a base element. The description of a role includes two things. Firstly, it specifies the type of the elements allowed in the set. Secondly, it uses a *navigation expression* to describe the elements in the set, in relation to the base element. Therefore, the power of the roles mechanism depends on the expressiveness of the navigation language used. The navigation language used in Cumbia will be presented in section 4.2.3.

To illustrate what a role is, consider the simplified BPMN metamodel depicted in figure 4.5. This figure only shows types of elements, the inheritance hierarchy formed by these types, and the relations that can be established between instances of those types. We are now going to describe a role in this metamodel, `[TASKS IN PROCESS]`, that serves to refer to the elements of type `Task` that are enclosed in a `Process`. The description of this role first specifies that the relevant elements are found by following the relation *flowElements* of `Process`, and then filtering by the type of the element, which must be `Task`[3].

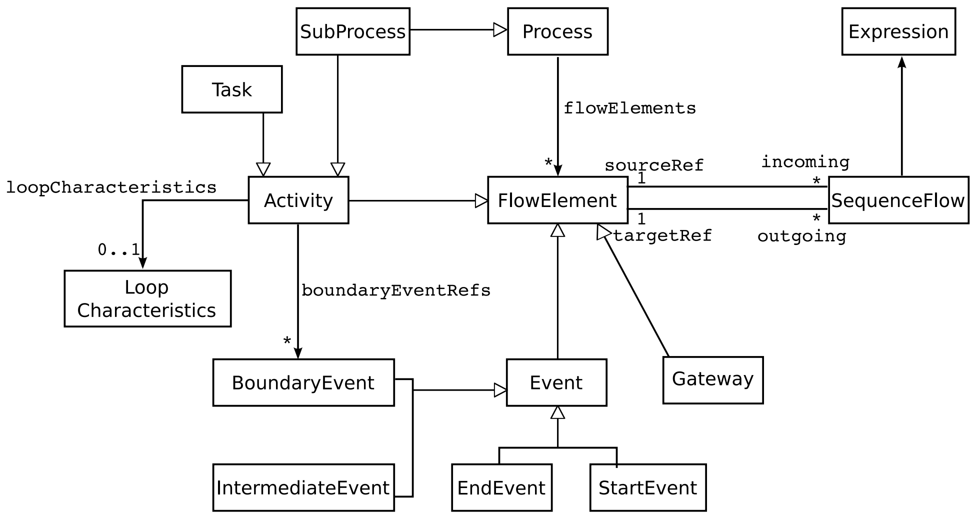


Figure 4.5 Simplfied BPMN metamodel

Applying the definition of the role to the BPMN process depicted in figure 4.6, we can identify the following: with respect to the top-level process, the role `[TASKS IN PROCESS]` is fulfilled by two tasks, namely Query Shipment State and Alert Responsible; with respect to the sub-process, the role `[TASKS IN PROCESS]` is fulfilled by the other four tasks. Note that without the restriction by type, the sub-process would also be included in the role for the top-level process, because a `SubProcess` *is a* `FlowElement` as well.

The final matter about roles and their usage to specify event sources has to do with the time of evaluation of navigation expressions. In the case of models that

[3]The specification of BPMN 2.0 defines the relation *flowElements* between a `FlowElementContainer` and a `FlowElement`. In this specification, a `Process` *is a* `FlowElementContainer`, and a `Task` *is an* `Activity`, which in turn *is a* `FlowElement` [Obj09a]. The metamodel in figure 4.5 has been simplified with respect to the full BPMN metamodel.

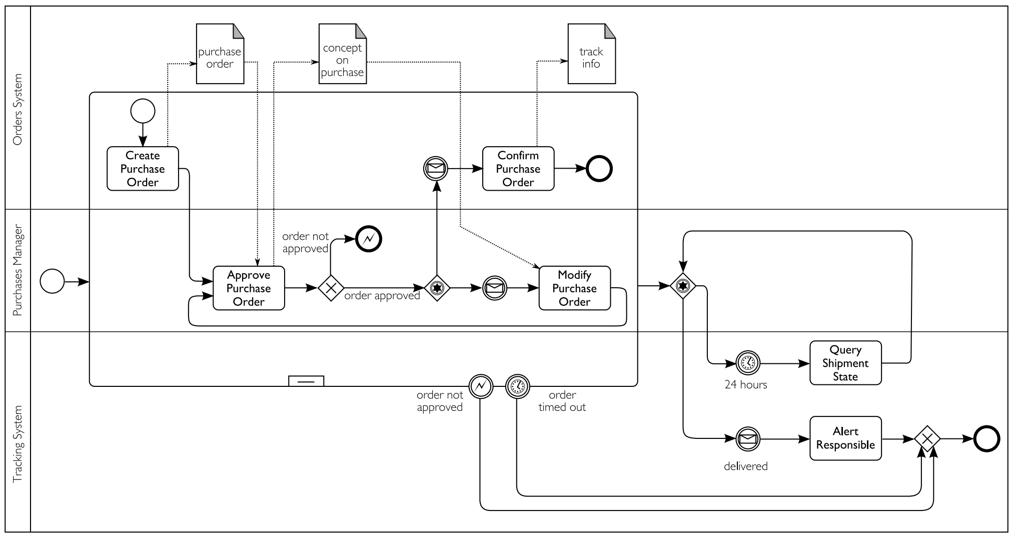


Figure 4.6 Sample BPMN process

do not change at run time, this evaluation can be done at the time of creating the instance of the model. However, if the model allows dynamic adaptation, then roles have to be recalculated for every change to the structure of the instance. For instance, in the previous example, if a `Task` were added to the top-level process, it should also be part of the role `[TASKS IN PROCESS]` for that process.

We will now analyze three cases that serve to describe and illustrate the two interaction mechanisms. We have used some elements from a metamodel whose execution model is based on token passing.

Case 1: Interaction between an entity and its state machine.

A state machine externalizes the state of the entity; therefore, at run time the state machine has to be in sync with the entity. This synchronization is achieved using an interaction mechanism based on events. In this case, the entity generates events which are received by the state machine.

Figure 4.7 illustrates this situation with a concrete scenario. In this figure, the entity of an open object called `Task` has an attribute `hasToken`; also, it has a method `setToken()` that makes the attribute true. The state machine of this open object has two states, namely *Waiting* and *Active*, and two transitions called *activation* and *deactivation* that connect the two states. The state machine and the entity are consistent only if one of the following statements is true: *a*) the state machine is in state *Waiting* and `hasToken` is false; or *b*) the state machine is in state *Active* and `hasToken` is true.

The event that triggers the transition *activation* is described with the expression `[ME]tokenReceived`. This means that the transition is triggered only when the same `Task` that owns the state machine (`[ME]`[4]) generates an event of type `tokenReceived`. In this case, the event is generated explicitly inside the method `setToken()` (see listing 4.3).

Finally, there is an action associated with the transition *activation*. This action, which invokes the method `activate()` on the entity, is executed each

[4]The role `[ME]` is special as it is available in every metamodel and always refers to the element that owns the state machine. `[ME]` can be considered to be equivalent to `this` in Java.

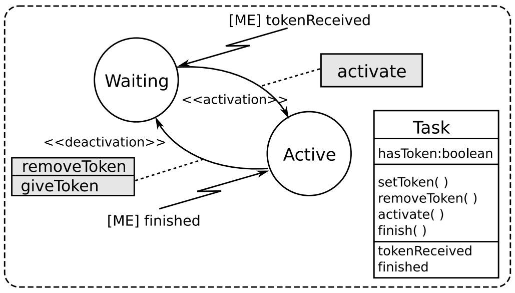


Figure 4.7 Coordination of the entity and the state machine of `Task`

Listing 4.3: Extract of the method `setToken()` in the entity of `Task`

```
public void setToken( )
{
    ...
    hasToken = true;
    ...
    // The method generateEvent( typeOfEvent ) is implemented in the class
        OpenObject, which is part of the framework provided.
    generateEvent( "tokenReceived" );
    ...
}
```

time this transition is triggered.

Case 2: Interaction between open objects using actions.

This second case involves more than one open object and uses an interaction mechanism based on method calls. For illustration purposes, the case considers the situation where an element of type `Task` is followed by an element of type `Gateway` in a process. According to the semantics of the language, after the task has been executed, its token should be passed to the gateway located after it.

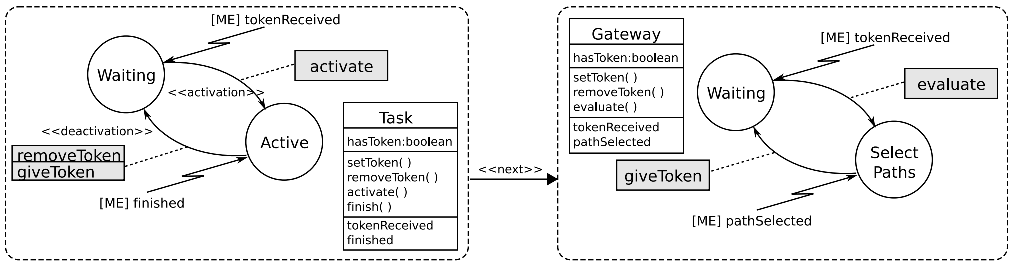


Figure 4.8 Interaction between `Task` and `Gateway`

In this particular case, the synchronization between `Task` and `Gateway` is achieved with *actions* associated to the transition *deactivation* of `Task`. This transition is triggered when the task finishes its execution and generates the event `finished` (the method `finish()` invokes the generation of the event). The transition also has two actions associated: the first one, *removeToken*, removes the token from the task, by calling the method `removeToken()` in the entity; the second action, *giveToken*, gives the token to the gateway by invoking its method `setToken()`. To do so, it is necessary to follow the association called *«next»* between the `Task` and the `Gateway`.

Listing 4.4 shows the implementation of the action *giveToken*. In this snippet, the parameter `event` corresponds to the notification of the reception of the event `finished`; the parameter `element` is a reference to the entity of the open object that received that event. The method `getElement( String )` is used to navigate between elements of a model, using the relations declared in the metamodel (see section 4.2.4).

Listing 4.4: Implementation of action *giveToken*

```java
public class GiveToken implements IAction
{
  public void execute( EventNotification event, Transition transition,
      IOpenObject element )
  {
    IGateway g = (IGateway) element.getElement("next");
    g.setToken( );
  }
}
```

Case 3: Interaction between open objects using event notifications.

The third case also involves two open objects, but uses an interaction mechanism based on events. Figure 4.9 shows a fraction of the state machines of two elements of the language, namely `Process` and `End Event`. In this language, the beginning and the end of a process are marked by `Start Events` and `End Events`: a `Process` has to terminate whenever one of its `End Events` receives a token.

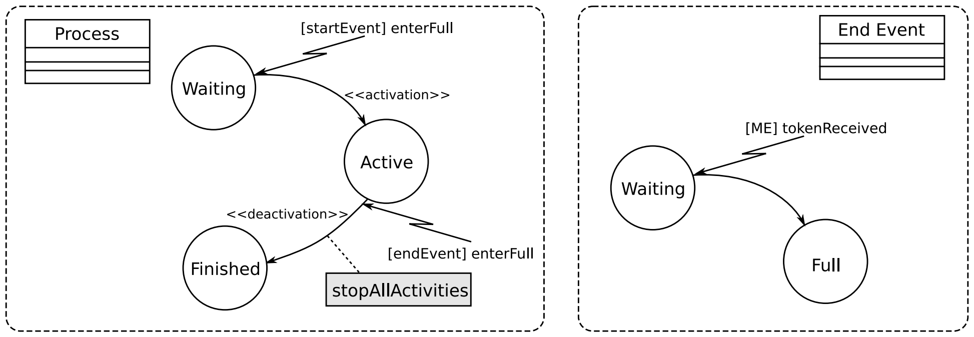


Figure 4.9 State machines of *a*) `Process` and *b*) `End Event`

The *deactivation* transition of the process' state machine in figure 4.9a, has an associated event described as `[endEvent]enterFull`. This means that the transition has to be taken whenever an element that fulfills the role `[endEvent]` generates an event of type `enterFull`. In this language, the role `[endEvent]` describes the `End Events` that are associated to each `Process` [5].

This case is different from the previous ones for three reasons. Firstly, the state machine is expecting an event generated by *another* open object. In the previous examples the role was always `[ME]`. Secondly, if the process has more

[5]In this metamodel, the role `[endEvent]` is associated to the element `Process`, and is restricted to elements of type `End Event`. Furthermore, the navigation expression associated to the role is `#self.endEvents` (see section 4.2.3).

than one `End Event`, then there is more than one possible source for the event. This means that the `Process` does not care who the exact generator of the event is, as long as it fulfills the role `[endEvent]`. Thirdly, the event in this case is not generated by the entity of an open object; instead, it is a notification of a change in a state machine: the event `enterFull` is generated when a transition that targets the state *Full* is taken.

Analysis of the interaction mechanisms

The interaction mechanisms presented are complementary and can be compared from three points of view. In the first place, method invocation is a synchronous mechanism: when a method is invoked, it is executed immediately and the caller cannot proceed until the execution of the method has finished. On the contrary, events are processed asynchronously: they are stored on reception, and they are processed at a later stage. However, it is not known in which order they are going to be processed. Having the two alternatives (synchronous and asynchronous interaction) is a positive feature, because each mechanism is adequate for certain situations. For instance, method invocation may be used in the points of a process where branches are synchronized (Joins), while event passing may be better for points where branches are created (Splits).

Another important difference between events and method invocation is the scope of the interaction. On the one hand, method invocation always involves one source and one target. On the contrary, event passing is more similar to a broadcast and the recipients of an event notification cannot be scoped. The relevance of this difference can be illustrated in a control-flow language that includes `OR-Splits` with conditions, such as YAWL (see the top part of figure 4.10). Using method invocation, the `Split` itself can evaluate each condition and activate only some of the branches (figure 4.10a). Conversely, if event passing is used, then all the ensuing tasks (T1, T2, and T3) receive the event and they have to evaluate the conditions (figure 4.10b).

The other criterion to compare the interaction mechanisms is coupling. With method invocations open objects are more coupled because of interface and structure dependencies. For instance, in case 2 the action *giveToken* knows that there is a *next* gateway, and that it has to invoke the method `setToken()` (see figure 4.8). Conversely, in case 3 the `Process` only knows that there is an element that can generate the event `enterFull`; the `Process` is oblivious about the implementation of the `End Event`, or its location, or its cardinality. In a similar way, the `End Event` is not aware of who is interested in receiving the events it produces.

4.2.3 Navigation language

In every metamodeling platform, it is important to be able to locate elements in a model. The most basic means to do so is with the identifier of each element. However, in many situations more powerful mechanisms are required.

In the context of programming languages, and particularly in the context of Java, various *navigation languages* (or *expression languages*) have been developed. Using those languages it is possible to write expressions to evaluate in the context of an object graph. The result of this evaluation is usually a collection of objects. Some of the best known navigation languages for Java are the Object-Graph Navigation Language – OGNL [Dav04], the JSP Expression Lan-

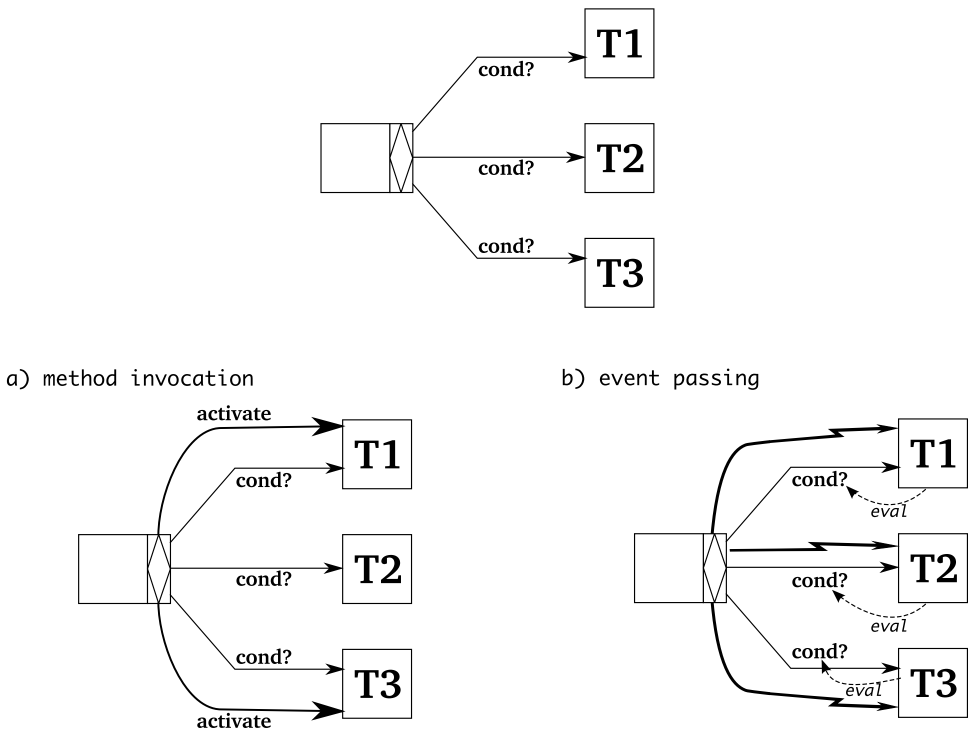


Figure 4.10 Differences between *a*) method invocation and *b*) event passing

guage [ABB+05], and MVEL [The06]. Furthermore, GPath [KGK+07], which is integrated with Groovy, can also be used to navigate and select Java objects.

Those languages offer roughly the same set of basic characteristics, like navigating object graphs using properties. In particular, they offer powerful means to navigate properties where the cardinality is larger than 1. Furthermore, they can also invoke methods of those objects and evaluate or compare their results against other properties, against other methods' results, or against some constants. On the other hand, some of these languages offer more 'advanced' functionalities such as the usage of variables and the definition of functions. Unfortunately, these 'advanced' functionalities have turned these navigation languages into scripting languages. Thus, their implementation and adaptation is more complex than the implementation of a single navigation language.

In order to navigate models based on open objects, we designed a navigation language inspired on the aforementioned languages. The characteristics of this language are all related to navigating the models, and they were heavily influenced by the necessity of describing the roles. Furthermore, the implementation of the language is not applicable to Java objects because it was built to be used specifically with open objects.

To present the navigation language, we will first show some examples, and then we will describe the main characteristic of the language. The following navigation expressions are valid in the context of the simplified BPMN metamodel shown in figure 4.5. They depend on the elements described in this metamodel (`Process`, `FlowElement`, `SubProcess`, etc.), and on the relations established between them (*flowElements*, *boundaryEventRefs*). Furthermore, they rely on some attributes which are not shown in the figure but are very important for BPMN semantics (e.g. elements in BPMN can be marked as loops by setting an attribute called `loopCharacteristics`). In this metamodel the root element of a model is always of type `Process`.

Sample navigation expressions

1. All the elements of the root process:

   ```
   #root.flowElements
   ```

2. All the elements inside either the root process or one of the subprocesses, depending on the context of evaluation:

   ```
   #self.flowElements
   ```

3. All the subprocesses contained in the root process:

   ```
   #root.flowElements.{? #this isOfType "SubProcess"}
   ```

4. All the boundary events associated to elements in the root process:

   ```
   #root.flowElements.boundaryEventRefs
   ```

5. All the boundary events associated to elements in the root process, and in its subprocesses, at every depth:

   ```
   #root.flowElements.
         <! #this.boundaryEventRefs ->
            #this.{ (#this isOfType "SubProcess") ?
                    (#this.flowElements) : (nil)}>
   ```

6. The members of the root process that are loops and can be executed more than 10 times:

   ```
   #root.flowElements.
         {? isNotNil? #this.loopCharacteristics &&
           #this.loopCharacteristics::loopMaximum( ) > 10 }
   ```

7. The first two outgoing flows of an element:

   ```
   #self.outgoing[0-1]
   ```

We now present the main characteristics of the navigation language, which can be seen in the previous examples. Afterwards, we discuss two of these examples in more detail. The specification of this language is presented in appendix C.

- The navigation language can be used to write expressions of two kinds: *paths* describe sets of elements[6], while *value expressions* describe the result of invoking a method or evaluating a property on an element described by a path.

  In the examples, the expression `#root.flowElements` is a *path* and it describes the set of elements in a `Process`. Conversely, the *value expression* `#this.loopCharacteristics::loopMaximum()` is used to invoke the method `loopMaximum()` in an object located with a path.

[6]More precisely, ordered sets. In a path, the elements are organized according to some criteria, and there are no repeated elements.

- Expressions can be absolute or relative. Absolute expressions start with the keyword `#root`, which refers to the root element of a model. The only thing required in the context to evaluate an absolute expression is a model instance.

- Relative expressions start with the keyword `#self` and are relative to an element that must be provided as the context to evaluate the expression.

  For example, if the `Process` at the root of the model is used as context to evaluate the expression `#self.flowElements`, the expression produces the same results as `#root.flowElements`. Instead, if another process is passed as context to evaluate the expression, the set of elements obtained with `#self.flowElements` will be the set of elements inside the passed process.

- Expressions starting with `#this` are also relative but they have to be evaluated in the context of another navigation expression.

- The language can be used to navigate relations between elements, and find the elements that match each subexpression. For instance, the expression `#root.flowElements.boundaryEventRefs` is evaluated in four steps.

  1. First, the only element in the collection is the root of the model (`#root`).
  2. Then, the only element currently in the collection (the root of the model) is removed, and all the elements associated as 'flowElements' are added to the collection.
  3. Then, the elements currently in the collection are removed, and all the elements associated to them as 'boundaryEventRefs' are added to the collection.
  4. The last step is returning the collection assembled.

- It is possible to filter the elements added to a path based on their position in a relation (for relations of type 'sequence') or based on the key (for relations of type 'map').

  For example, the expression `#self.outgoing[0-1]` can be evaluated on an element where the relation *outgoing* is defined. The result of evaluating this expression is a set with at most 2 elements, which are the first two outgoing `SequenceFlows` of the element used as context.

- It is also possible to evaluate conditions using the elements currently in the path. Conditions can be evaluated with respect to the types of the elements, their properties, the result of invoking methods on them, or constants. Also, it is possible to compose conditions by using boolean operations.

  The expression `#root.flowElements.{? #this isOfType "SubProcess"}` shows an example of this. In the first place, this expression selects the elements associated as 'flowElements' to the root of the model. Then, these elements are filtered by evaluating their type, and only those that are of type `SubProcess` are kept on the collection.

- It is possible to add elements to the path in a recursive way (this is illustrated in one of the following examples).

The previous examples presented the basic characteristics of the language. We now present two additional examples which provide a further illustration of the characteristics of the language.

Example 1 – The members of the root process that are loops and can be executed more than 10 times

```
#root.flowElements.
      {? isNotNil? #this.loopCharacteristics &&
        #this.loopCharacteristics::loopMaximum( ) > 10 }
```

This path uses a conditional expression and finds all the members of the root process which are marked as loops[7] and can be executed more than 10 times. The function `isNotNil?` verifies if there is an element that matches the expression. The second part of the condition invokes the method `loopMaximum( )` on the result of the path `#this.loopCharacteristics` and then verifies if it is larger than 10. In order to invoke a method it is necessary to have a path that describes 0 or 1 elements. It is a run time error to invoke a method using a path that can return more than one element.

Example 2 – The boundary events associated to elements in the root process, and in its subprocess, at every depth

```
#root.flowElements.
      <  #this.{ (#this isOfType "SubProcess") ?
                        (#this.flowElements) : (nil)}
        ->
         #this.boundaryEventRefs >
```

This expression not only selects events attached to elements contained directly in the root process, but also looks for events attached to elements in subprocesses. To achieve this, this expression uses the capabilities for recursion of the navigation language. We will now analyze part by part this expression.

A recursive expression is described with a structure of the form:

```
< Conditional Expression -> Path >
```

Furthermore, a 'Conditional Expression' has a structure of the form:

```
(condition) ? ( alternativeTrue ) : ( alternativeFalse)
```

The semantics of this works as follows.

- The recursive expression is evaluated in a context, which is a collection of elements. In this example, the recursive expression is evaluated in the context of all the members of the root process. This is the result of the expression `#root.flowElements`.
- Evaluating a recursive expression involves maintaining two collection of elements: one represents the result of the expression; the other contains the elements in the context that still have to be analyzed.
- The conditional expression is used to evaluate the elements in the context. The 'condition' is evaluated for each element in the context and, depending on its result, the elements in 'alternativeTrue' or 'alternativeFalse' are added to the context to be later evaluated.

[7]In BPMN this is achieved by setting the attribute loopCharacteristics.

In the example, the conditional expression is the following:

```
{(#this isOfType "SubProcess") ? (#this.flowElements) : (nil)}
```

In this case, there is a conditional expression over the type of `#this` which has two possible outcomes. If `#this` is indeed a `SubProcess`, the members of this `SubProcess` are added to the context and are thus marked to be explored. If `#this` is not a `SubProcess`, then there is no recursion, as 'alternativeFalse' is `nil`.

- The other side of the recursion expression indicates which elements have to be used to assemble the final result. In this case, the expression

```
#this.boundaryEventRefs
```

indicates that the final path returned by the expression should include the boundary events connected to all the elements evaluated.

Listing 4.5: Java code equivalent to the recursive navigation expression

```java
SortedSet<BoundaryEvent> getBoundaryEventsRecursively(Process root)
{
  SortedSet<BoundaryEvent> result = new TreeSet<BoundaryEvent>( );
  SortedSet<FlowNode> toCheck = root.getFlowElements( );
  Collection<FlowNode> checked = new TreeSet<FlowNode>( );

  getBoundaryEvents(result, toCheck, checked);

  return result;
}

void getBoundaryEvents(SortedSet<BoundaryEvent> result,
  SortedSet<FlowNode> toCheck, Collection<FlowNode> checked)
{
  if (toCheck.isEmpty( ))
    return;

  FlowNode nThis = toCheck.first( );

  result.addAll(nThis.getBoundaryEventRefs( ));
  if (nThis.isOfType("SubProcess"))
  {
    List<FlowNode> subElements = nThis.getFlowElements( );
    for(FlowNode nThis2 : subElements)
    {
      if (!checked.contains(nThis2))
      {
        toCheck.addAll(nThis2.getFlowElements( ));
      }
    }
  }

  checked.add(nThis);
  toCheck.remove(nThis);

  getBoundaryEvents(result, toCheck, checked);
}
```

The recursive expression analyzed is equivalent to the Java program shown in listing 4.5. This code shows two important aspects of the semantics of the language. On the one hand, using `SortedSets` for `result` and `toCheck` ensures that elements are not repeated inside those collections, and that their order is maintained. On the other hand, the `checked` collection and the `contains(nThis2)` verification serve to ensure termination. In this case it is not required because the structure of the metamodel does not allow circular references (a subprocess cannot be contained in itself), but these verifications are always present when recursive expressions are used.

4.2.4 Open object definition

The specification and implementation of an open object entails four parts:

- The description of the general characteristics of the new element.
- The description of the state machine structure.
- The implementation of the entity
- The implementation of the actions associated to its state machine.

In this section we illustrate these steps using a concrete example. It is important to note that the example is a base type, i.e. a type that is not an extension of another type of the metamodel. In section 4.2.5 we will show how new open objects are created on top of existing open objects.

Structure of an open object definition

The first part that we will analyze is the declaration of the general characteristics of the open object. In the Cumbia platform, this is done with an XML file with a structure as exemplified by listing 4.6[8]. The Cumbia Kernel is capable of interpreting the information provided in this file to create instances of this open object. Furthermore, the Cumbia Kernel uses this information to perform various consistency checks on the metamodels and on the models.

The definition of an open object can be divided in four sections: attributes of the open object; relations with other open objects; information about the events generated by the entity; and the roles relevant to the open object.

The attributes of an open object are the following:

- **Name:** Each open object must have an unique name within a metamodel.
- **Entity Class:** This is the name of the class for the entity of the open object.
- **Interface:** This is the name of the interface for the entity. The entity class must implement the interface.
- **State machine:** This is the name of the state machine of the open object.
- **Abstract:** It is possible to mark open objects as abstract. It is not possible to create instances of abstract open objects, and thus they have to be extended in order to be used.

[8]Appendix B.1 presents an XSD schema that formalizes this structure.

Listing 4.6: Structure of an open object definition

```
<type name="Activity"
      entityClass="sample.elements.Activity"
      interface="sample.elements.IActivity"
      statemachine="defaultActivity">

  <!-- Relations -->
  <relation name="parentProcess" cardinality="simple"
            relationType="simple" targetTypeName="Process"/>
  <relation name="workspace" cardinality ="simple"
                   relationType="simple" targetTypeName="Workspace"/>
  ...

  <!-- Generated Events -->
  <event name="activated" />
  <event name="activityAbort" />
  <event name="executionEnded" />
  ...

  <!-- Roles Definition -->
  <role name="PROCESS" description="Parent process">
    <role_detail type="Process">#self.parentProcess</role_detail>
  </role>
  <role name="WORKSPACE"
        description="The workspace running inside the activity">
    <role_detail type="Workspace">#self.workspace</role_detail>
  </role>
  ...

</type>
```

The second section of the description specifies the relations that the open object has with other elements of the metamodel. Each relation is described with the following parameters:

- **Name:** Each relation is named and this name must be unique within the open object.
- **Cardinality:** This can either be '`simple`', to indicate a '0..1' cardinality, or '`multiple`', to indicate a '0..*' cardinality. No other cardinalities are currently supported by the metamodel.
- **Relation type:** For relations with '`simple`' cardinality, the relation type is always simple, because there may be at most one reference to an other open object. For relations with '`multiple`' cardinality the relation can be implemented with a '`sequence`' or with a '`map`'. In the first case, the references are ordered. In the second case, there must be a key for each reference.
- **Target type name:** This is the super-type of the elements in the other end of the relation.

This information about relations is used to verify that models are conformant to the restrictions of the metamodel (see section 4.3). There are 4 verifications performed for each relation established between two elements in a model. These verifications are performed when a model is loaded.

1. The relation must be declared for the source element, or one of its super-types

2. The type of the target element must be the target type declared or a subtype of it.
3. The compliance to the cardinality of the relation is verified.
4. For relations of type ‘map’, a key must be provided with the reference.

The third part of the description specifies the names of the events generated by the entity of this open object. The entity of the open object will be capable of invoking the generation only of the events declared in this section. Besides these events, an open object can also generate the events declared in the state machine description.

The fourth section is the description of the roles that are relevant for this open object. For each role, at least one navigation expression has to be provided. Each navigation expression also specifies the type of the elements expected to be found with the expression. Furthermore, this part establishes the mapping between the structure of the metamodel and the source events associated to transitions of the state machines.

Structure of a state machine definition

The definition of the structure of a state machine is also done with an XML file[9]. This file must have a structure such as the file depicted in listing 4.7. This description is structured around the states of the state machine, and for each one the transitions that start on it are described.

Listing 4.7: Structure of a state machine definition

```
<state_machine name="defaultActivity" initial_state="Inactive">

    <!-- State Active -->
    <state name="Active"
           enter_event="enterActive"
           exit_event="exitActive">
        <transition name="ActivityAborted" successor="Aborting">
            <source_event source_name="ME" event_name="activityAbort" />
            <before_event name="beforeActivityAborted" />
            <after_event name="afterActivityAborted" />
            <actions>
                <action name="abortWS"
                        class="sample.actions.activity.AbortWS" />
                 ...
            </actions>
        </transition>
        ...
  </state>
  ...
</state_machine>
```

For each state the following basic information is provided:

- **Name:** Each state in a state machine must have a unique name.
- **Enter Event / Exit Event:** The name of the events generated when the state machine reaches the state, and when the state machine leaves the state.

[9]In chapter 6 we present a graphical editor to define the state machines and generate the corresponding XML files. Appendix B.2 presents an XSD schema that formalizes this structure.

Furthermore, for each state a set of transitions are described. The commonality between those transitions is that all of them originate from the same state. The following are the attributes that must be provided for each transition:

- **Name:** Each transition in a state machine must have a unique name.
- **Successor:** The name of the state targeted by the transition.
- **Source Event:** The event that triggers the transition. This description must include a role name (that must match one of the roles described in the open object, or in a super-type) and the name of an event. When a metamodel is loaded by the Cumbia Kernel, the types declared for the roles are used to verify if these events can be generated.
- **Before Event / After Event:** The name of the events generated when the transition is taken (before executing the actions, and after executing the actions).

Finally, each transition can have associated actions. Each action must have a unique name (unique within the transition), and the name of the class that implements the action. Furthermore, the order in which actions are defined is relevant because they are executed in the order specified.

Implementation of the entity

Inside the Cumbia platform, the implementation of the entity of an open object does not have to be done from scratch because the open objects framework offers many base functionalities. Thus, the implementation of an entity can focus on the behavior that is relevant to the domain. In order to have access to these existing functionalities, the entity must be an extension to the class `OpenObject`.

Listing 4.8 shows a fragment of the implementation of the entity of the open object called `Activity`. The two methods shown (`abortActivity()` and `stop()`) are declared in the interface `IActivity`.

The implementation of `abortActivity()` shows how to invoke the generation of events, using the method `generateEvent(String eventName)`.

On the other hand, the implementation of `stop()` shows how another element of the model is found by following the relation ‘workspace’ which was declared in the description of the open object. The method `getElement(String relationName)` is also implemented in the class `OpenObject`.

Figure 4.11 shows the structure of the main classes and interfaces in the open objects’ framework[10]. This hierarchy is structured in five levels, and each level has different responsibilities.

- **Level 1 – Navigable Elements:** these elements define the interface and implement the behavior to navigate object graphs. The relations between open objects are supported at this level.
- **Level 2 – Metamodel platform:** these elements implement the basic functionalities of a metamodeling framework. At this level, it is possible to have metamodels and models, although their elements do not have to be open objects.

[10]Appendix A presents all the details about these interfaces.

Listing 4.8: Extract of the implementation of the entity of an open object

```java
public class Activity extends OpenObject implements IActivity
{
  ...
  public void abortActivity( )
  {
    generateEvent( "activityAbort" );
  }

  public void stop( ) throws CumbiaException
  {
    IWorkspace workspace = ( IWorkspace )getElement( "workspace" );
    workspace.stop( );
    ...
  }
  ...
}
```

- **Level 3 – Open objects platform:** at this level, the metamodeling platform is more specific and its elements are either open objects, or are elements that can interact with open objects. For instance, the event passing mechanisms are implemented at this level.

- **Level 4 – Open objects:** these elements implement the complete structure and semantics of the open objects, using what was defined in the previous levels. For instance, the classes to support the state machines are implemented at this level.

- **Level 5 – Metamodel element:** in this level of the hierarchy are implemented the elements used in specific metamodels. In the case depicted in the figure, at this level we have the entity of `Activity` and its interface.

The classes discussed and included in the figure are just a subset of the classes defined in each layer of the Cumbia Kernel. Besides these classes there are many others that implement related functionalities. For instance, the classes to load metamodel and model definitions are implemented at the *metamodel platform* level. The implementation of those classes is refined in levels 3 and 4, in order to load open object specific information.

Implementation of the actions

Each action associated to a transition is implemented in a different class. This class must implement the interface IAction which is defined in the open object kernel, and only declares the method `execute()` (see listing 4.9).

The parameters in the method are the context to execute the action. These parameters provide the following information:

- **Event:** This is information about the event notification that triggered the transition. Using this information, it is possible to have access to the element that generated the event.

- **Transition:** This is the transition where the action is installed.

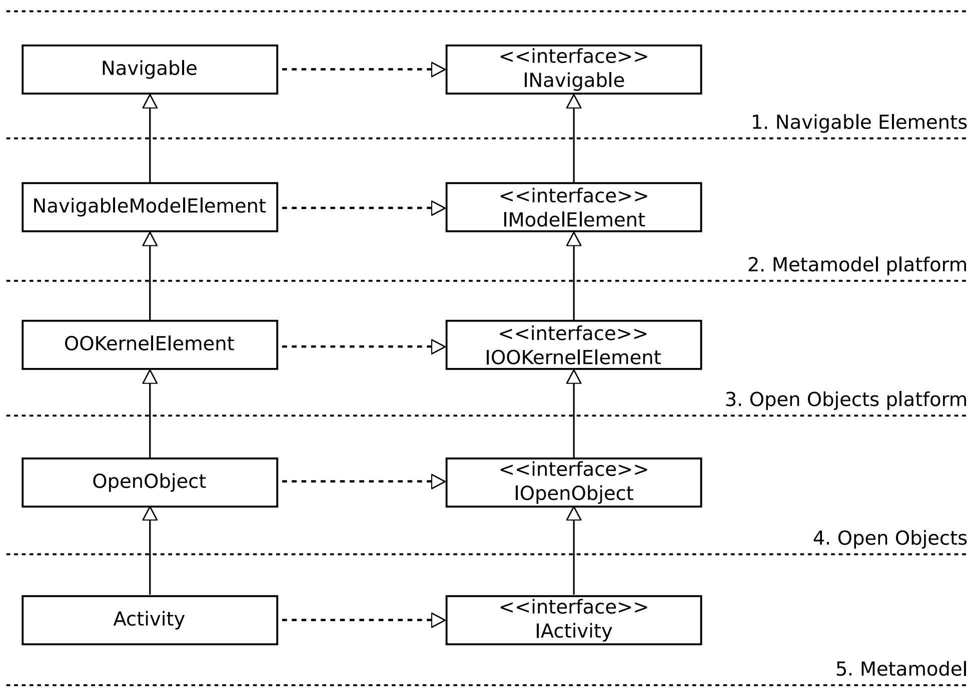


Figure 4.11 Basic classes and interfaces in the open object's framework

- **Element:** This is the element that owns the state machine with the transition. Using this reference it is possible to query the state of the element, invoke one of its methods, and also navigate the model. For instance, in the sample code the action finds the *Workspace* of the `Activity` and invokes its method `abort()`.

Listing 4.9: Implementation of an action

```
public class AbortWS implements IAction
{
  public void execute( EventNotification event, Transition transition,
      IOpenObject element )
  {
    // Implementation of the action
    IAtomicActivity activity = ( IAtomicActivity )element;
    IWorkspace workspace =
      ( IWorkspace )activity.getElement( "workspace" );
    workspace.abort( );
  }
}
```

4.2.5 Extension mechanisms

In metamodeling platforms, the relation of inheritance or extension between metaclasses serves two purposes. On the one hand, it contributes to structuring the metamodel. For instance, in class-based object oriented frameworks, hierarchies of classes are created with inheritance relations, and these relations are used to handle polymorphism. On the other hand, inheritance can be used to define new metaclasses as extensions to existing ones, instead of doing it from scratch. Therefore, extension is also a reuse mechanism.

The specific semantics of extension relations vary for each platform. As an example, consider class inheritance in C++ and in Java. Although they share many similarities, there are also important differences that have to be considered when using those languages. In particular, the contrast between static method binding in C++ and dynamic method binding in Java, is a clear manifestation of the different semantics of the inheritance relation [Cra07].

Since open objects define several additional elements with respect to standard objects, we had to specify new semantics for the extension relation. Moreover, since open objects are not composed of a single part, we refined the extension relation to differentiate between extensions to the state machine and extensions to the entity. In the rest of the section we will discuss the different types of extensions that can be used to define new open objects, which can be combined to a certain degree. In section 4.8 we will illustrate these extensions in a concrete metamodel.

The following are the five types of extensions that can be used to create new open objects. These can be applied in combination to define new extended elements with greater flexibility. All the characteristics that are not explicitly modified, are inherited by the extended elements.

- Modifications to the entity description.
- Modifications to the relations.
- New implementation of the entity.
- Modifications to the state machine.
- New state machine.

These five types of extensions, as well as their subdivisions form the hierarchy depicted in figure 4.12.

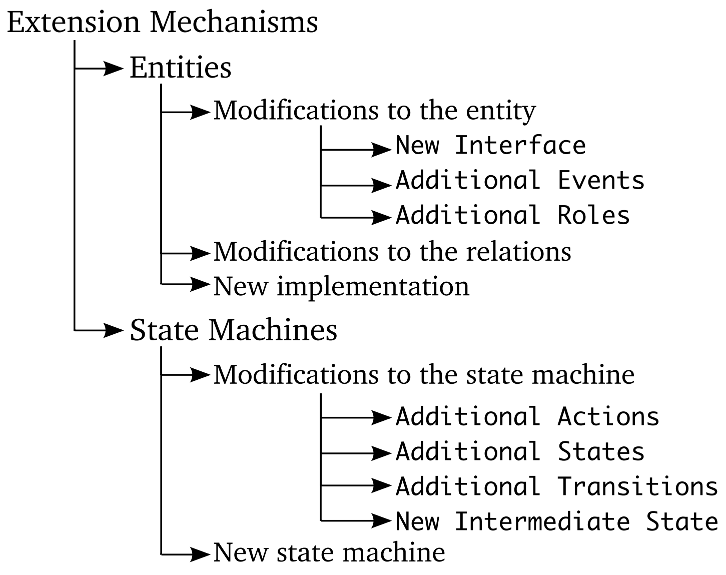


Figure 4.12 Hierarchy of extensions

Modifications to the entity description

The first type of changes only affects the details of the entity that are provided in the description of the open objects. These changes can be further divided in three kinds.

1. New interface

The first possible change to the description of an entity is to change its interface. This does not affect any of the other aspects of an open object or its entity: it does not affect the events generated, the roles available, or the state machine. The motivation for changing the interface is usually to provide new services. Therefore, changing the interface of the entity usually implies changing the class that implements that interface.

A change of interface only requires the name of the new interface that the entity must implement (see listing 4.10).

Listing 4.10: Extension example: new interface

```
<extended_type  name="ExtendedElement"  extends="BaseElement">
  <new_entity interface="cumbia.samples.IExtendedInterface" >
  ...
  </new_entity>
</extended_type>
```

2. Additional events

Another way of extending the entity of an open object is by declaring extra events that it can generate. This is usually accompanied by a change in the implementation of the entity, which triggers the generation of these new events.

Listing 4.11 shows how an extended element is described to generate additional events with respect to the base element.

Listing 4.11: Extension example: additional events

```
<extended_type  name="ExtendedElement"  extends="BaseElement">
  <new_entity>
    <new_event name="additionalEvent1" />
    <new_event name="additionalEvent2" />
    ...
  </new_entity>
</extended_type>
```

3. Additional roles

The third extension to an entity adds new roles that can be of relevance to its state machine. As for base elements, the definition of additional roles requires the type of elements, and some navigation expressions to find the elements that belong in the role. This is illustrated in listing 4.12.

Modifications to the relations

Another way of extending an element is by adding new relations. Listing 4.13 shows how this is done to create an `ExtendedElement` that extends the `BaseElement` by adding a new relation with the element `OldElement`.

This kind of extension is frequently accompanied by changes to the entity's implementation, and also by the definition of new roles based on the new relations.

Listing 4.12: Extension example: additional roles

```
<extended_type  name="ExtendedElement"  extends="BaseElement">
  ...
  <new_role name="NEW_ROLE" >
      <role_detail type="OldElement">
       ... <!-- Navigation expression to find the elements -->
      </role_detail>
      ...
  </new_role>
  ...
</extended_type>
```

Listing 4.13: Extension example: new relations

```
<extended_type  name="ExtendedElement"  extends="BaseElement">
  <new_relation name="connections"      cardinality="multiple"
                relationType="sequence" targetTypeName="OldElement"/>
   ...
</extended_type>
```

New implementation of the entity

When this kind of extension is applied, a new entity implementation is used in the extended open object. A possible motivation for this is a change to the description or to the interface of the entity, as described before. Nevertheless, this extension mechanism can also be applied to specialize or to modify the behavior of the entity.

The most frequent situation we have found to apply this extension mechanism is to create specialized tasks or activities. Generally, workflow specification languages include generic activities that have to be configured somehow to perform specific actions (see figure 4.13a). For instance, in order to use `Invoke` nodes in BPEL, they need to be configured with the information of a *Partner Link*. The problem with this is that the semantics of the nodes is limited by the configuration system. For instance, in BPEL it is not possible to configure an `Invoke` node to choose among similar service providers based on external criteria.

The alternative to a configuration approach is to have a hierarchy of elements that perform specific tasks. For instance, in YAWL a *decomposition* has to be specified for each Task appearing in a process. The platform provides several of those decompositions, and additional ones can be introduced by end users (see figure 4.13b).

If YAWL decompositions are modeled with open objects, their specializations are naturally extensions where the implementations of entities have been replaced. Considering that the base element (`Decomposition`) has a method `start( )` that is invoked by a `Task` when it is activated, then each extension to `Decomposition` should implement this method in a different way.

The following code snippet (listing 4.14) is the concrete way to define such an extended element in a metamodel. As it can be seen, a new element is defined (`ManualDecomposition`), and the only parameter specified is the name of the class that implements its entity. For all the other aspects, this new element is similar to an existing element called `Decomposition`.

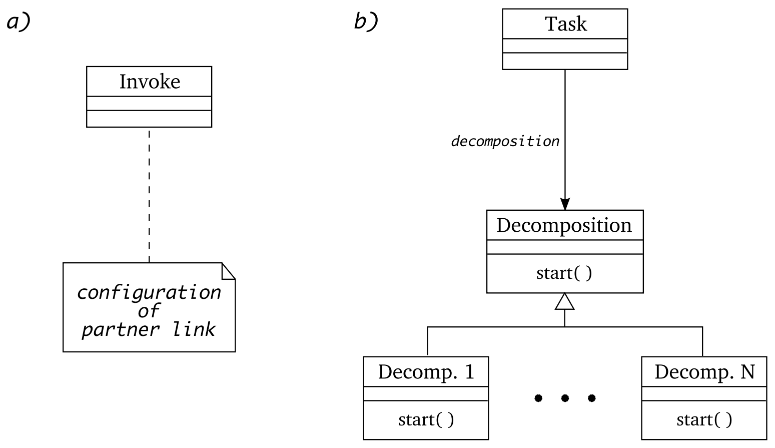


Figure 4.13 a) The configuration approach (BPEL) vs. b) the specialization approach (YAWL).

Listing 4.14: Extension example: new implementation of the entity.

```
<extended_type extends="Decomposition" name="ManualDecomposition">
  <new_entity
      entityClass="cumbia.yawl.decompositions.ManualDecomposition"/>
</extended_type>
```

Modifications to the state machine

The third kind of extensions applicable to the open objects involve changes to the structure of the state machines. However, some changes to the state machines are more likely than others to introduce consistency problems in a metamodel. Therefore, we decided to support only four kinds of changes, which add additional elements to the state machine, and which have a small possibility of creating inconsistencies. On the contrary, changes that remove elements from the state machine, or that change its structure, are very likely to introduce consistency problems.

As an example of this, consider the simple scenario depicted in figure 4.14. In this scenario, initially there are two kinds of open objects: `Light` is an entity that can be either *On* or *Off*; and `Control` represents a switch that has two states, namely *Deactivated* and *Activated*, which are connected with the transitions *activation* and *deactivation*. `Light` and `Control` interact when the state machine of `Control` transitions to the state *Activated*. At that point, the state machine of one or more `Lights` go from *Off* to *On*. The procedure is analogous when the `Control` is deactivated.

Now, we are going to consider an extension to `Control` that is only a `Button`. This extension was created by changing both the structure of the state machine of `Control` and its entity. Nevertheless, the idea is that `Button` is a particular kind of `Control`. This button should be pressed to turn the lights on, and it should be pressed again to turn the lights off. However, the button is not aware of the state of the lights, and therefore it only has one state. The problem in this case is that `Light` is expecting the event `enterActivated`, but if the control is a `Button`, the event is never going to be generated. If `Light` were expecting the events `afterActivation` and `afterDeactivation`, the inconsistency would still

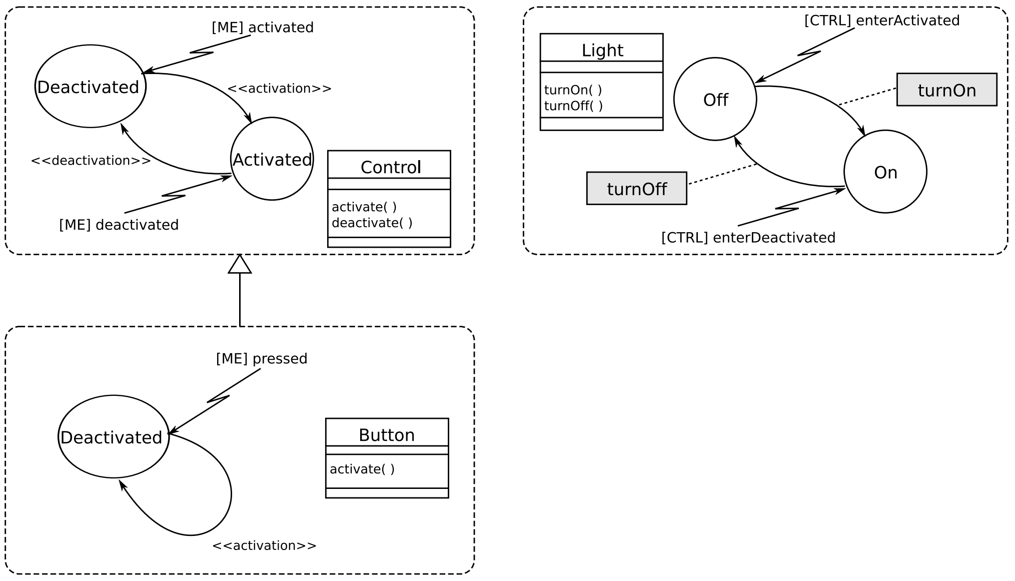


Figure 4.14 Inconsistency scenario

be present.

As we mentioned before, open objects support four categories of changes to the structures of the state machines. These categories were selected because they conserve most of the structure of the state machines, and thus they are *less likely* to introduce inconsistencies. Even with this reduced set of changes supported, it is not possible to guarantee the consistency.

The types of changes supported are the following:

1. Additional actions
2. Additional transitions
3. Additional states
4. New intermediate states

When a new extended element which modifies the state machine is created, several of the described categories of changes can be combined. The following listing (listing 4.15) shows the basic structure to define such an extended element. Inside the element `<state_machine_extensions>` several changes can be described, and they are applied in order to the base state machine. If the provided categories of changes are insufficient, the state machine can be replaced altogether.

1. Additional actions

This category of changes to a state machine simply adds new actions to existing transitions. These new actions can only be added at the end of the sequence of actions associated to a transition.

Listing 4.16 shows how two new actions are associated to transition *transitionToExtend* in the element `ExtendedElement`. For each new action to add, it is necessary to specify its name and the name of the class that implements it.

2. Additional transitions

Listing 4.15: Empty state machine extension

```
<extended_type name="ExtendedElement" extends="BaseElement">
  <state_machine_extensions>
    ...
    <! Extensions to the state machine -->
    ...
  </state_machine_extensions>
</extended_type>
```

Listing 4.16: Extension example: additional action

```
<extended_type name="ExtendedElement" extends="BaseElement">
  <state_machine_extensions>
    ...
    <!-- Add Actions to transition -->
    <add_actions transitionName="transitionToExtend">
      <action name="actionName1"
              class="cumbia.samples.extension.Action1" />
      <action name="actionName2"
              class="cumbia.smaples.extension.Action2" />
    </add_actions>
    ...
  </state_machine_extensions>
</extended_type>
```

This category of changes to a state machine adds additional transitions between existing states. It is also possible to add a new transition that starts and ends in the same state.

To add a new transition it is necessary to specify the same details that are used when a transition is defined in a base state machine. These details include: the name of the transition; the states connected with the new transition; the event that triggers the transition; the events that are generated when the transition is taken (before and after executing its activities); and the details about any action associated to the new transition.

Listing 4.17 adds a new transition to the state machine of `ExtendedElement`.

Listing 4.17: Extension example: additional transition

```
<extended_type name="ExtendedElement" extends="BaseElement">
  <state_machine_extensions>
    ...
    <!-- Add Transition -->
    <add_transition name="newTransition"
                    source_state="state1" successor="state2">
      <source_event source_name="ME" event_name="triggerNewTrans" />
      <before_event name="beforeNewTransition" />
      <after_event  name="afterNewTransition" />
      <actions>
        ...
      </actions>
    </add_transition>
  ...
  </state_machine_extensions>
</extended_type>
```

3. Additional states

This category of changes adds additional states to an existing state machine. Since state machines are connected graphs, adding a state also requires the addition of at least one *incoming* transition that starts from an existing state: in this context, incoming transitions have as destination the new state, while outgoing transitions start from the new state.

As shown in listing 4.18, the definition of a new state requires some basic details about the new state and also the information about all the incoming and outgoing transitions. The information about the new state includes its name, the events associated to it, and whether this will be the initial state for the state machine.

Listing 4.18: Extension example: new state

```
<extended_type name="ExtendedElement" extends="BaseElement">
  <state_machine_extensions>
    ...
    <!-- Add State -->
    <add_state name="newStateName"
               enter_event="enterNewState"
               exit_event="exitNewState"
               initial_state="false">
      <incoming_transitions>
        <transition name="newTransitionIn" source_state="state1">
          ...
        </transition>
        ...
      </incoming_transitions>
      <!-- Outgoing transitions are optional -->
      <outgoing_transitions>
         <transition name="newTransitionOut" successor="state1">
           ...
         </transition>
         ...
      </outgoing_transitions>
    </add_state>
    ...
  </state_machine_extensions>
</extended_type>
```

4. New intermediate states

The fourth category of changes supported adds intermediate states. The difference between this kind of changes and the preceding ones is that in this case the new state is connected to an existing transition. Figure 4.15 illustrates the two possible ways to do this. In the original situation, there is a transition called *fromAtoB* that connects the states *A* and *B*. In the situation where the new state is added *before* the existing transition, a new transition called *newTr* is used to connect states *A* and *NewState*. Additionally, the existing transition *fromAtoB* is reconfigured to connect *NewState* and *B*. The organization of the new and the old transition is reversed when the new state is added *after* the existing transition.

It should be noted that when an intermediate state is added all the existing events are kept. The only thing that changes is that intermediate events are added.

The following snippet of code shows how an intermediate state is added *after* an existing transition. In this case, information is provided about the new state and about the new transition.

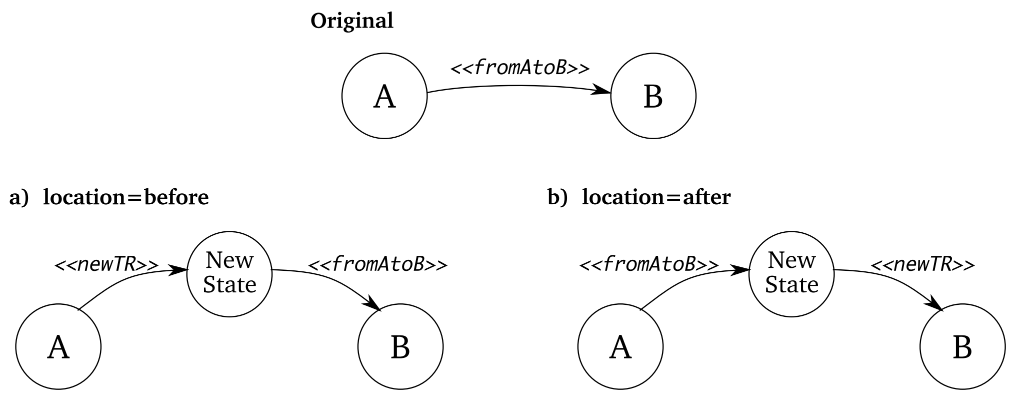


Figure 4.15 New intermediate state possibilities: located a) before, and b) after

Listing 4.19: Extension example: new intermediate state

```
<extended_type name="ExtendedWorkspace" extends="Workspace">
  <state_machine_extensions>
    ...
      <!-- Add Intermediate State -->
      <add_intermediate_state transitionName="fromAtoB" location="after">
        <additional_state name="NewState"
                          enter_event="enterNS" exit_event="exitNS">
          <additional_transition name="newTr">
    ...
          <!-- Details about the transition -->
          ...
          </additional_transition>
        </additional_state>
      </add_intermediate_state>
    ...
  </state_machine_extensions>
</extended_type>
```

New state machine

The last kind of extension mechanism entirely replaces the state machine of the base element. This is done when the structure of the state machine of the extended element is incompatible with the state machine of the base element. The drawback of this, is that it is easier to introduce inconsistencies in a metamodel when this mechanism is used.

As an example, consider the life cycle of a Decomposition in YAWL (figure 4.16): every work item has to be offered, allocated and started before being completed. However, this life cycle is not compatible with automatic activities, that do not need to be offered nor allocated. Therefore, a different state machine is required.

The following is the concrete way to define such an extended element in a metamodel (listing 4.20). A new element is defined (`AutomaticDecomposition`), and the only parameter specified is the file with the description of the state machine. For all the other aspects, this new element is similar to an existing element called `Decomposition`.

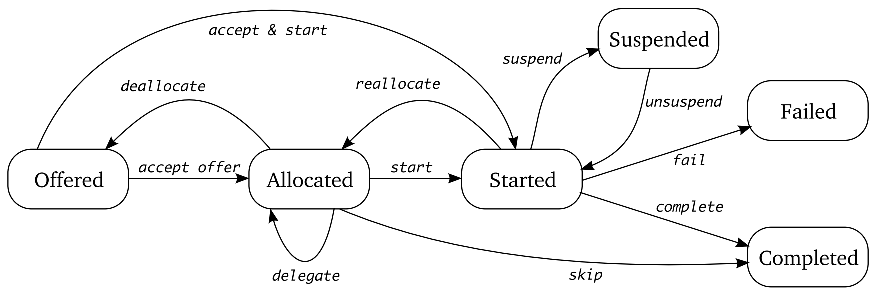


Figure 4.16 Life cycle of a `Work Item` in YAWL [The08]

Listing 4.20: Extension example: new state machine.

```
<extended_type name="AutomaticDecomposition" extends="Decomposition">
  <new_state_machine name="adSM" file="AutoDecSM.xml" />
</extended_type>
```

4.2.6 Other perspectives on open objects

Open objects have similarities to other works and proposals, both with respect to the general goals sought, and to the strategies and tactics utilized. We will now discuss some of these works, and we will focus on the main similarities and differences.

APEL is a graphical and enactable formalism, and also a platform for building process support systems [DEA98, EVLA$^+$03]. There are several key ideas that Cumbia adopted, which were first developed and tested in APEL. One of them is the usage of state machines or, in APEL's terminology, state diagrams - SD. In APEL, the SDs describe the evolution of entities over time, and the events and conditions that cause state changes. This is conceptually very similar to what the state machines of open objects achieve, and their differences are mainly of a technical nature. For example, while open objects use roles and the types of events to filter them, in APEL events are broadcasted and rules are used to decide if a given state change should occur. Other differences can be found in the events generated. For example, in Cumbia a temporal signal does not exist; supporting it would require the development of a special kind of open object.

A central part of APEL is a base metamodel that defines a series of concepts that are common to all process based applications. This base metamodel can be extended to support specialized applications, using mechanisms that are similar to those found in the open objects. For example, a new element can be created as an extension to an existing one and inherit its SD. If necessary, subtypes can be introduce modifications to the SD, but it is only allowed to add new states or transitions. In chapter 5 we provide a little more information about APEL, and about the strategy to extend the base process metamodel by means of composing multiple domains.

In their book about *metaobject protocols*, Kiczales, des Rivières, and Bobrow [KdRB91] discuss and illustrate an idea for enabling users to create their own languages, or modify existing ones. This is achieved by means of interfaces to the language implementations, which they call metaobject protocols. This approach results in *open* languages, that are not completely fixed, and can be adapted to create variants that suit specific situations. This objective is extremely close to the goal in Cumbia of supporting very expressive languages based on concepts of particular domains.

We can consider that the open objects specification defines a metaobject protocol, and that the Cumbia Kernel and the open objects framework implement this specification. Since they are structured following an object oriented approach, the parallel to the metaobject protocol presented in [KdRB91] is even closer. The basic design and implementation of the open objects defines a family of languages, where every language can be obtained by applying extension mechanisms to refine or extend the behavior of existing open objects. In the previous section, we have shown that many of those extension mechanisms are based on the key object-oriented concepts of subclassing and specialization, just as is suggested in [KdRB91].

An important concept in the design of metaobject protocols if that of *metaobjects*, which are reifications of the base elements of the languages. In the case of object-oriented languages, these metaobjects include *classes* and *methods*. In the case of open objects, the relevant metaobjects are the open objects themselves and their constituent parts (entities, state machines, and actions). Currently, further decompositions are not possible, i.e. in the metaobject protocol for open objects is not possible to treat the methods of an entity on par with the actions of a state machine. The execution of models based on open objects is the main responsibility of the Cumbia Kernel, and it is able to do so while oblivious of the specific behavior defined in the metamodel. This is possible because the Cumbia Kernel only operates in terms of the metaobjects previously mentioned.

Reflection, is a technique frequently used with metaobject protocols, and it is also a technique used extensively in the Cumbia Kernel. The main idea behind the usage of reflection in both cases is that it serves to open up language implementation, while restricting the elements that can be effectively accessed. This strategy rises the level of abstraction, and hides unnecessary implementation details.

The characteristics of the open objects and of the Cumbia Kernel, and also the ways to define new languages, makes it possible to draw some relations between Cumbia and works on extensible interpreters. As in the case of Cumbia, the goal of extensible interpreters is to support extensible languages. According to [Sch71], there are two alternative approaches to implement an extensible interpreter. In the first one, the interpreter offers a set of primitive instructions, which implement some basic semantics. Language implementations provide instructions with concrete implementations based on the primitives provided. Later on, when programs written with the language are executed, the interpreter has to find and execute the sets of primitives to execute, given the instructions in the program and the behavior associated to each one in the language implementation.

The outlined approach results in interpreters that provide only a set of low level instructions and where all languages are implemented using the same one. Therefore, when new languages are defined it does not matter if they are base languages or if they extend existing ones, because they end up being defined in terms of the same primitives. This creates performance problems, as it introduces an overhead on the interpretation [Sch71].

The second approach of [Sch71] to implement extensible interpreters results in a different strategy to handle language extensions. In this approach, the set of base instructions is not fixed, but instead grows with the languages. Therefore, the definition of an extended language produces an extended interpreter, whose instructions reflect the operations in the extended language, and are at the same level as the primitives originally provided.

Depending on the point of view, we can see that both approaches are present in Cumbia. If the Cumbia Kernel and the operations of the open objects are considered as the basis of the interpreter, the view presented is very close to the first approach: the basic set of instructions never changes and the elements of each language are always described using the same terms.

Conversely, if we analyze Cumbia from the stand point of the languages implemented on top of it, we can see a picture that is closer to the second approach of [Sch71]: if we build a new language on top of an existing one, the elements of the new language are treated in the same exact way as the elements of the base one. This property is maintained even when language extensions are accumulated.

State machines are a central element in the open objects proposal, and they are used for two purposes: one is to represent the state of the elements, and the other is to coordinate their execution. In other proposals, state machines fulfill different roles, and, in particular, they are used for verification and analysis purposes. For instance, ensuring the correct interoperation of the elements in a component based architecture is a task that can only provide limited results if based on the static analysis of component interfaces. On the contrary, doing an analysis of the behavior and interaction between components [BCP07, BBS06], when the behavior is specified by means of state machines, is likely to provide more useful information.

STSLib is another approach to model component based systems, and especially, to model their types, behaviors, and interactions [FR08, FLR08]. According to its documentation, STSLib offers four kinds of functionalities: firstly, it supports the formal design of software components using symbolic transition systems (STS) to achieve this goal; then it supports their verification using STSs; the third point is the enactment of the component models, for which STSLib offers an execution platform; finally, STSLib is also capable of interfacing with external tools, such as model-checkers.

Several parallels can be traced between elements of Cumbia and elements of STSLib, and especially with respect to the enactment of component models. One clear parallel is that primitive elements are typed, which is also a core characteristic of open objects. Furthermore, composites in STSLib relate primitive components, and thus fulfill a role similar to that of metamodels in Cumbia. Primitive components are also composed of two complementing aspects: on the one hand, there is the *data part* of the component, which corresponds to the entity of open objects; and on the other hand, there is the *protocol* which attains the same goals as state machines in open objects.

4.3 Metamodels and model definition

This section has two parts. The first one shows how a metamodel is constructed from the definition of the open objects used in it. Then, the second part shows how models conformant to a given metamodel (or an extension to it) are built.

4.3.1 Metamodel definition

The definition of a new metamodel is basically the union of the open objects included in it. Listing 4.21 shows the basic structure of a metamodel definition, which can be divided in four main parts[11].

The first part only indicates the name and the version of the metamodel. These details are important to assess the compatibility of the models.

The second part of the metamodel description has references to the files with the specifications of the state machines. This information about the state machines is external to the specification of the open objects to make it possible for elements to share their state machines.

The third part of the file is the central one, as it contains the descriptions of the open objects included in the metamodel. These open objects can be base types or extended types. The process of loading a metamodel performs the following checks to assess the consistency of the information described in this file.

- Check that the information required for each element type is complete.
- Check that names that should not be duplicated, are not duplicated (names of element types, states within a state machine, relations in an element type, etc.).
- Check that relations are properly defined (the type and cardinality are valid, and the type appears in the metamodel).
- Check that extended elements extend types declared in the metamodel.
- Check that the roles that appear in the state machines are declared in the metamodel.
- Check that roles are properly defined. This include checking that the types appear in the metamodel and that navigation expressions are well formed and match relations in the metamodels.
- Check that the events expected by the state machines match with the events generated by the types and super-types that can assume a role.

The final part of a metamodel definition is the configuration of the *runtime controller* of the metamodel. The runtime controller is a component required by the kernel that implements functional requirements related to the execution of the models. There is a generic runtime controller that provides a basic implementation of those operations, but they can be specialized for a metamodel if it has specific requirements. A metamodel definition must select an implementation of the runtime controller, by providing the name of the class, as well as additional configuration information. All the details about the functionalities of the runtime controller are discussed in section 4.4.

4.3.2 Model definition

The definition of models is done with an XML file, which can be divided in four main parts, as shown in listing 4.22[12]. The structure of the files to define the

[11]Appendix B.1 presents an XSD schema that formalizes this structure.

[12]Appendix B.3 presents an XSD schema that formalizes this structure

Listing 4.21: Basic structure of a metamodel definition

```
<!-- 1. Metamodel definition -->
<metamodel name="XPM" version="1.0">

  <!-- 2. Reference to state machine specifications -->
  <state_machine_reference name="defaultXPMNode" file="xpmNodeV3.xml"/>
  <state_machine_reference name="defaultProcess" file="processV3.xml"/>
  <state_machine_reference name="defaultWorkspace"
                           file="workspaceV3.xml" />
  ...

  <!-- 3. Open objects included in the metamodel -->
  <!-- XPMNode -->
  <type name="XPMNode"
        entityClass="uniandes.cumbia.xpm.elements.XPMNode"
        interface="uniandes.cumbia.xpm.elements.IXPMNode"
        statemachine="defaultXPMNode"
        abstract="true">
    ...
  </type>

  <!-- Process -->
  <extended_type name="Process" extends="XPMNode">
    ...
  </extended_type>

  <!-- Worspace -->
  <type name="Workspace"
        entityClass="uniandes.cumbia.xpm.elements.workspace.Workspace"
        interface="uniandes.cumbia.xpm.elements.workspace.IWorkspace"
        statemachine="defaultWorkspace">
           ...
  </type>
  ...

  <!-- 4. Runtime controller configuration -->
  <runtime class="uniandes.cumbia.xpm.runtime.XPMRuntime">
      <memory>
        <data name="admin" type="String">admin</data>
      </memory>
  </runtime>
</metamodel>
```

models does not vary with the metamodel. Therefore, it is not necessary to develop a new language and a new parser for each workflow language supported.

As with metamodel definitions, the first part of a model definition provides general information about the model. In this case, this information includes the name of the model, and the name and version of the metamodel that it should be conformant to.

The second part of a model definition describes extensions to the metamodel that are required for that specific model, and only for that specific model.

The third part of the model definition provides additional configuration for the runtime controller that is to be used with that model.

Finally, the fourth and most important part describes the structure of the model, which includes its elements (e.g. the activities of a process) and the way in which they have to be related (e.g. with flows between them).

Listing 4.22: Basic structure of a model definition

```
<!-- 1. Model definition -->
<definition modelName="Event Registration" metamodel="XPM" version="1.0">

    <!-- 2. Extensions to the metamodel -->
    <metamodel_extensions>
        ...
    </metamodel_extensions>

    <!-- 3. Runtime controller configuration  -->
    <runtime class="...">
        ...
    </runtime>

    <!-- 4. Model Structure -->
    <model_structure root="MainProcess">
        <elements>
        ...
        </elements>

        <connections>
        ...
        </connections>
    </model_structure >
</definition>
```

Metamodel extension

In many cases, the elements included in a metamodel are not sufficiently specialized to build concrete workflows. For instance, in the case of BPMN the element called `Task` represents any activity that is part of a business process. Among others, `Tasks` serve to represent manual activities performed by humans, activities that transform data, and automatic activities that interact with external applications. It is clear that an implementation of the BPMN metamodel cannot provide an implementation of `Task` that can fulfill all the required responsibilities. Therefore, in order to have executable BPMN models, the BPMN metamodel has to be extended to provide adequate specializations of the element `Task`. Each one of those will be capable of performing one of the type of atomic activities required in that particular business process[13]. For example, one specialization could serve to consume remote services using the SOAP protocol, another one could serve to transform the data obtained from that service, and a third one could serve to require some human input through some external application.

In the case of Cumbia, for each model a metamodel extension can be provided which builds over the base metamodel declared in the attribute of the model. Since the elements in a metamodel extension are expected to be be very specific to the metamodel, the specification of this extension is included in the file that contains the description of the model. The definition of an extended metamodel, which is shown in listing 4.23, uses the same mechanisms described in section 4.2.5 to describe the extended types. In the example provided, three elements are added to the metamodel that was used to describe the process (`XPM` according to listing 4.22). The elements in the sample ex-

[13]In some metamodels the elements are detailed enough and thus do not need to be specialized, but only configured. For example, a metamodel for WS-BPEL should include from the beginning configurable elements to invoke SOAP web services synchronously and asynchronously, and elements to transform data using XSLT.

tended metamodel are called `AutoProcess`, `StoreInformationWorkspace`, and `UploadInformationWorkspace`.

Listing 4.23: Definition of model-specific metamodel

```xml
<!-- 2. Extensions to the metamodel -->
<metamodel_extensions>
  <!-- An extended  process -->
  <extended_type extends="Process" name="AutoProcess">
    <state_machine_extensions>
      <add_actions transitionName="StartA">
        <action
         class="uniandes.cumbia.xpm.tests.ActionAssignProcessResponsible"
         name="registerProcessResponsible"/>
      </add_actions>
    </state_machine_extensions>
  </extended_type>

  <!-- Ad-hoc workspace StoreInformationWorkspace -->
  <extended_type extends="Workspace" name="StoreInformationWorkspace">
    <new_entity
     entityClass="uniandes.cumbia.xpm.tests.StoreInformationWorkspace"/>
  </extended_type>

  <!-- Ad-hoc workspace UploadInformationWorkspace -->
  <extended_type extends="Workspace" name="UploadInformationWorkspace">
    <new_entity
      entityClass="uniandes.cumbia.xpm.tests.UploadInformationWorkspace"/>
  </extended_type>

</metamodel_extensions>
```

Runtime controller configuration

For each model, it is also possible to provide additional configuration details for the Runtime Controller to use. For example, listing 4.24 shows how to configure the controller to enable its instances to be debugged. Additionally, each model can also use an implementation of the runtime controller different from the one provided by the metamodel. For simplicity in the implementation, the configuration of a runtime controller uses the same memory structure that is used in metamodel elements.

Listing 4.24: Runtime controller configuration

```xml
<runtime>
  <memory>
    <data name="debug" type="String">true</data>
  </memory>
</runtime>
```

Model structure

The last part of a model definition is the specification of its structure, which must be conformant to the specified metamodel and to the extended metamodel. The structure of a model is specified in two parts: first the elements used in it are declared, and then the connections between them are specified.

Listing 4.25: Definition of the elements of a model

```
<!-- 4. Model Structure -->
<model_structure root="MainProcess">
  <elements>
    <element name="MainProcess" typeName="AutoProcess" />

    <element name="StoreInformation"
             typeName="AutoActivity" />
    <element name="StoreInformationWS"
             typeName="StoreInformationWorkspace"/>

    <element name="UploadInformation"
             typeName="AutoActivity" />
    <element name="UploadInformationWS"
             typeName="UploadInformationWorkspace">
      <memory>
        <data name="url" type="String">
          http://localhost/uploadInfo.php
        </data>
      </memory>
    </element>
    ...
  </elements>

  <connections>
    ...
  </connections>
</model_structure>
```

The declaration of the elements appearing in a model is exemplified in listing 4.25. As it is shown, each element in the model has a name which has to be unique in the model. The type of each element is also specified, and this type name must match with the name of a type in the metamodel or in its extension.

It is also possible to configure the elements by providing data to include in their memory[14]. In this way, when instances of the model will be created, the elements will have that information loaded in their memories. In the example presented, only the element called 'UploadInformationWS' requires a special configuration. This element is of type `UploadInformationWorkspace`, which is an extension to the type `Workspace` and was declared in the extended metamodel. Furthermore, from listing 4.25 it results that this type is configured by putting in its memory a data element of type String with the name "url".

The second part of a model structure declaration is the specification of the associations established between the elements previously described. These associations are based on the relations[15] declared for each type and we call them *connections*.

Listing 4.26 shows sample connections between elements in the model. The following are the important details abut these connections.

- For each connection, the *sourceElement* and the *targetElement* must be elements declared in the model.
- The name of the relation in each connection (*relationName*) must match the name of a relation declared in the type of the source element. The

[14]The characteristics and the rationale of the memory of open objects was described in section 4.2.1.

[15]See section 4.2.1 for information about relations and its valid attributes.

Listing 4.26: Definition of connections between elements of a model

```
<model_structure root="MainProcess">
  <elements>
    ...
  </elements>

  <connections>
    <!-- Activity StoreInformation -->
    <connection relationName="activities" sourceElement="MainProcess"
                targetElement="StoreInformation"/>
    <connection relationName="parentProcess"
                sourceElement="StoreInformation"
                targetElement="MainProcess"/>
    <!-- Workspace of StoreInformation -->
    <connection relationName="workspace" sourceElement="StoreInformation"
                targetElement="StoreInformationWS"/>
    <connection relationName="activity" sourceElement="StoreInformationWS"
                targetElement="StoreInformation"/>

    <!-- Activity UploadInformation -->
    <connection relationName="activities" sourceElement="MainProcess"
                targetElement="UploadInformation"/>
    <connection relationName="parentProcess"
                sourceElement="UploadInformation"
                targetElement="MainProcess"/>
    <!-- Workspace of UploadInformation -->
    <connection relationName="workspace" sourceElement="UploadInformation"
                targetElement="UploadInformationWS"/>
    <connection relationName="activity"
                sourceElement="UploadInformationWS"
                targetElement="UploadInformation"/>
    ...
  </connections>
</model_structure>
```

relation can also be declared for a super-type of the source element.

- The type of the target element must be compatible with the type declared for the relation. This means that the target type must be of the type declared, or of a sub-type.
- If a relation has the cardinality type 'multiple', then many connections with the same *sourceElement* and the same *relationName* may appear in the model. For instance, the relation 'activities' is defined as 'multiple' in the type `Process`. Therefore, it is valid to have several connections between *MainProcess* and other elements, using the relation 'activities'.
- The order in which connections are defined is relevant to relations of type 'sequence': in those cases, the order of the associations between the open objects' instances matches the order of the connections declared in the model specification.
- For relations of type 'map', the additional attribute *key* must be used to associate two open objects.

4.4 Model execution

This section analyses the execution of open objects based models. There are two main elements involved in this. On the one hand, there is the Cumbia Kernel whose main responsibility it is to coordinate the execution of the elements in a model. To do so, the Cumbia Kernel handles all the tasks related to generating and distributing events.

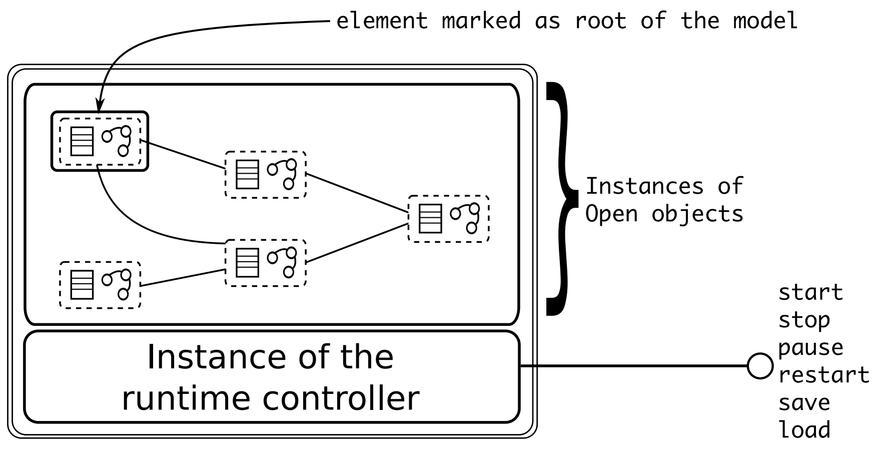

element marked as root of the model
Instances of Open objects
Instance of the runtime controller
start
stop
pause
restart
save
load

Figure 4.17 A model instance at run time

On the other hand, there is another element that intervenes in the execution of a model. We called it the *Runtime Controller*, and it can be considered part of the model instance because there is one instance of it for every model instance (see figure 4.17). The runtime controller has three main functions. Firstly, it handles certain operations that are common to every metamodel such as starting, stopping, pausing and restarting the execution of an instance, and also saving and loading the state of a running instance. In principle, the implementations of these operations are the same for every metamodel and are provided in the basic runtime controller. However, if a different implementation is required, a specialized runtime controller can be selected, as was shown in the previous section. As an example, consider the operation `pause`: in one workflow language, pausing a running instance may happen immediately and thus require every task to stop instantly. In another language the pause operation may wait until every active task finishes its execution normally. These differences in the behavior of the operations are implemented in specialized runtime controllers.

The second function of the runtime controller is to generate certain event notifications about the execution of the model instance. These events are associated to the aforementioned operations, and thus their generation is also metamodel dependent. The most important of those events are `started`, `stopped`, `paused`, `restarted`, `saved`, and `loaded`. Nevertheless, these events are not generated instantly as the corresponding operations are invoked. Instead, they are generated asynchronously and are often related to events generated by the elements in the model instance. For instance, the event `stopped` is usually associated to the event `enterStopped` or similar in the top level element of the model instance.

Finally, the third function of the runtime controller is controlling how events are processed. If necessary, the runtime controller can implement variable policies to manage the processing of events in a metamodel. In most situations this is not necessary and the events of every element in a model instance are processed with the same priority.

4.4.1 Model Instantiation

The first step towards executing a model is creating a model instance. In order to do so, it is necessary to have available all the information and resources of the metamodel, and the definition of the model. The following are the actions required to create a model instance.

Loading the metamodel

In the first place, the metamodel has to be loaded in the Cumbia Kernel. This loading process includes parsing and verifying the metamodel definition. Furthermore, when a metamodel is loaded, its associated resources are also located, in particular the classes used for the entities and the actions.

Loading the model

After loading the metamodel, the next step is to load the model. This step has two parts. First, if there is a metamodel extension, it has to be loaded and verified, in the same way as the base metamodel. Afterwards, the model definition is loaded and its structure is verified to check its conformance to the metamodel.

Instantiating the model elements

With all the information about the elements of the metamodel and the elements in the model definition, it is possible to create an instance of the model. This step basically requires creating instances of the open objects as declared in the model definition, initializing those instances, and then establishing the connections between them.

In the Cumbia platform, an instance of an open object has all its elements reified as objects. This includes the entity and the state machines with its parts, and also the actions. In this way, all these elements can be queried and manipulated as objects. The entity of the open object is an instance of the class specified in the metamodel definition. Therefore, it is a dynamically loaded class that is instantiated using the reflection API of the Java platform. Similarly, the classes for the actions associated to the state machine are also loaded and instantiated using the reflection mechanism. The object representations of the state machines are constructed to reflect the structure described in the metamodel.

After each open object and its components are instantiated, they are initialized. The most important aspect of this is giving to each element its unique name specified in the model definition. Furthermore, the state machine has to be put into the initial state declared in the specification of the open object. Finally, the memory of the element is initialized, with the information provided in the model definition.

Finally, the instances of the open objects are connected. This is done by creating the connections described in the model definition. For this, the Cumbia Kernel uses the methods defined in the interface INavigable (see figure 4.11). Additionally, the instances are also connected from the point of view of the events consumed and generated. This means that roles are evaluated, and that the open objects instances are registered as listeners of the necessary events.

Instantiating the runtime controller

The instantiation process is concluded by creating an instance of the runtime controller and associating it with the instances of the model elements. Furthermore, since the runtime controller can be configured, the last step is applying this configuration.

4.4.2 Model Execution

After a model instance has been created, the execution of its elements is controlled by the Cumbia Kernel. To do so, the Kernel controls the processing of events received by each open object. This includes updating the state of the state machines, executing the actions associated to the transitions, and distributing the events generated.

For each model instance, there is a single execution thread that processes events. The events received by each open object are stored in a queue until they are processed. The runtime controller has the responsibility of selecting which queue to process at any given moment. Therefore, the events in a queue are guaranteed to be processed in the order of arrival, but there is no guarantee about the order in which events received by different open objects are processed.

The first step to process an event is to check the current state of the state machine and the transitions that originate in that state. Then, the type of the event is compared against the type of the events that trigger those transitions. If there is a match, the element that generated the event is matched with the role declared in the transition. If these two tests are passed, the transition is selected to be taken. If no transition is selected, then the event is discarded. If there is more than one transition that matches the event received, any one can be selected. Note however that this situation is a design problem of the metamodel, which introduces uncertainty on the model execution.

When a transition is taken, the following actions occur (figure 4.18):

1. An event is generated to announce the departure from the previous state.
2. An event is generated to announce that the transition is starting.
3. The actions associated to the transition are executed.
4. An event is generated to announce that the transition was completed.
5. An event is generated to announce the arrival to the new state.

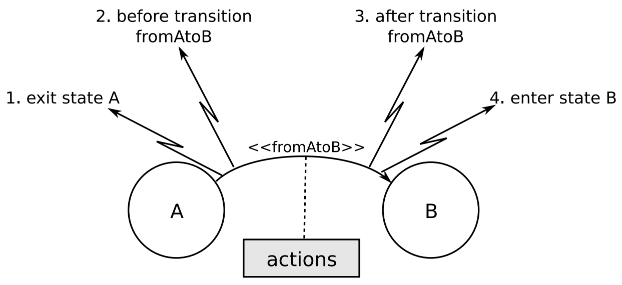


Figure 4.18 Events generated when a transition is triggered

The generation of those events is a responsibility of the Cumbia Kernel, which also distributes them to all the registered listeners. Each one of those, receives a copy of the original event notification, which is stored in the respective event queue to be processed.

The execution of actions is sequential and synchronous. Only when an action is marked as asynchronous, the Kernel does not wait until the action finishes in order to proceed with the execution of the following action. Furthermore, in the case of asynchronous actions, the actions are really executed in a different thread than the thread used to process the events. As it was previously said, the behavior of actions is not restricted. Therefore, an action can invoke methods of an entity and trigger the generation of new events and thus keep alive the execution of the model.

After all the actions associated to a transition have been executed, the last two events are generated. At this point, the current state of the state machine is updated, and a new event can be processed. This new event can be the one received by the same element as before, or it can be retrieved from a different queue. Selecting the queue to process is a responsibility of the runtime controller. However, the default implementation provided in the platform tries to be fair by processing events from different queues more or less evenly. This is achieved by maintaining a list that holds references to the queues that have not-yet processed events; after an event from a queue is processed, that queue is removed from the list (if it does not have other unprocessed events) or it is put at the end of the list.

As it was previously said, all the events in a model instance are processed in a single execution thread. This does not mean that there is no concurrency in the execution of a model instance. On the contrary, there are three basic means to achieve concurrency.

1. By using asynchronous actions. Since those actions are processed in different threads, everything they do, including invoking methods of the entities, is done in a thread different from the event-processing thread.

2. By spawning new threads in the implementation of the actions or of the entity methods. The main problem in this case is that the Kernel is not aware of the creation and management of those threads.

3. Concurrency can also appear when code external to the Kernel, and thus running in other threads, invokes methods of the entities. This can also be a problematic case, as the Kernel has no way to control these invocations and synchronize their execution with the other aspects of the execution. Therefore, it is advisable that the methods of the entities that can be invoked from outside fulfill one of these characteristics: either they only serve to query the state of the elements, or they interact with other elements by generating events.

An additional way of having concurrency is by using different threads to process the events. For instance, a different thread could be used to process the events of each queue, thus making the same guarantees about the order in which events are processed. Nevertheless, after various tests we realized that this strategy complicates the design of the metamodels and especially the synchronization of the state machines. Since the actions and methods of the entities can invoke methods of other elements, having multiple, unrelated threads increases the risk of race conditions.

On the contrary, using a single thread makes the possible sources of concurrency problems more visible and easier to detect. Since asynchronous actions are explicitly marked as such, then metamodel developers know that they should check them for concurrency problems and thus reduce the risk of encountering problems.

4.5 Workflow engines and the Cumbia Kernel

Now we will explain the general architecture of a workflow engine based on the Cumbia platform. This information is presented from a technical point of view, by identifying the parts that constitute an engine and their roles. In chapter 6 we present a development process that identifies the steps to develop these engines.

A workflow engine based on Cumbia can be seen as a component formed by the three elements shown in figure 4.19: the definition and resources of a metamodel, an instance of the Cumbia Kernel, and elements specific to the language supported by the metamodel. This architecture has similarities to the architecture of applications based on *embedded interpreters*, such as the one described in [Ram06]. In those applications, interpreters for scripting languages are used as libraries and can be easily replaced when the implemented language has to grow or change. In the case of Cumbia based engines this is possible because languages can be replaced by upgrading the Cumbia Kernel to work with other metamodels. This kind of changes should only require a reconfiguration of the Kernel to load the definition of the upgraded metamodels and their associated resources (state machines' definitions, implementations of entities and actions, and other complementary resources).

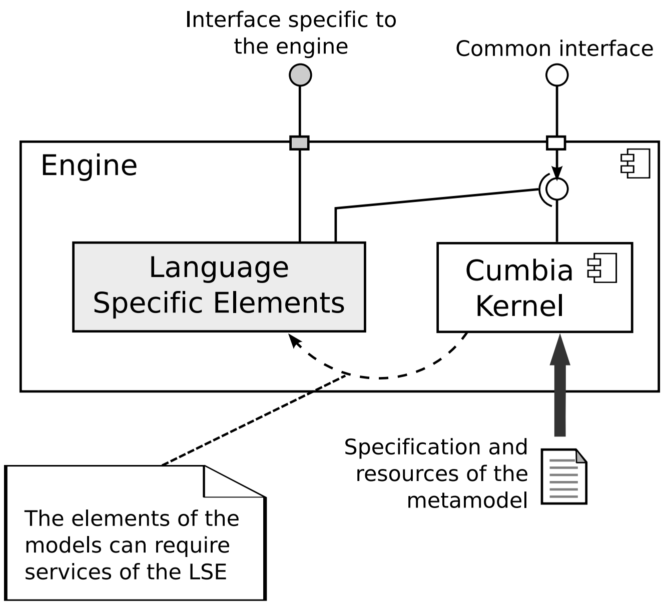

Interface specific to the engine
Common interface
Engine
Language Specific Elements
Cumbia Kernel
Specification and resources of the metamodel
The elements of the models can require services of the LSE

Figure 4.19 Basic architecture of a Cumbia based engine

The other core aspect of architectures based on embedded interpreters is permitting back and forth interactions between the interpreters and the host application: on the one hand, this means that the interpreter can make invocations on the host application as required by the *programs* interpreted; on the other hand, it means that the host application can also make invocations on the interpreter. The rest of this section describes the interactions between the Cumbia Kernel (the interpreter), and the other elements of a Cumbia-based engine (the host application).

As shown in the previous section, the Cumbia Kernel has a crucial role in the creation and execution of model instances, and offers an interface to allow the interaction with the elements in model instances. By using this interface, external elements can navigate the elements of a model instance, query the state and analyze the structure of their state machines, and also invoke any method on the entities of the open objects.

This interface also serves to analyze and modify the structure of the model. By offering this, it is possible to support workflow languages that require dynamic adaptation. Nevertheless, it should be considered that the support offered by the platform for this is very low level. Higher level concerns related to guaranteeing model consistency, or model and metamodel migration, are currently not provided by the Cumbia platform.

The functionalities provided by the Cumbia Kernel are expected to be shared by most workflow engines built on top of it. For example, it is expected that most engines will require run time information to persist, and therefore we implemented such functionalities in the Cumbia Kernel. However, languages and engines can also require particular functionalities that are not provided by the Kernel. There are basically two alternatives for the implementation of these functionalities.

The first alternative is to implement them as part of the metamodel, by distributing the code between its elements. For instance, in the case of BPEL it can be reasonable to implement the invocation of web-services in the code of the element `Invoke`. However, this alternative has a limitation when the functionalities affect many elements or have to be shared between several instances of a model, or even several model definitions. For example, BPEL engines are expected to expose a SOAP interface to query and control the processes, and to request the creation of new instances. This cannot be implemented in the meta-model, and thus it has to be implemented in something additional and external to the Cumbia Kernel.

The second alternative is to implement these functionalities in what figure 4.19 calls *language specific elements.* There are no restrictions on what these elements can be or do. For example, they can implement and expose additional interfaces to interact with the engine and with the model instances. Furthermore, they can also implement functionalities that may or may not interact with the Cumbia Kernel and with the model instances. On the other hand, the elements in the models can also use services offered by those specific elements.

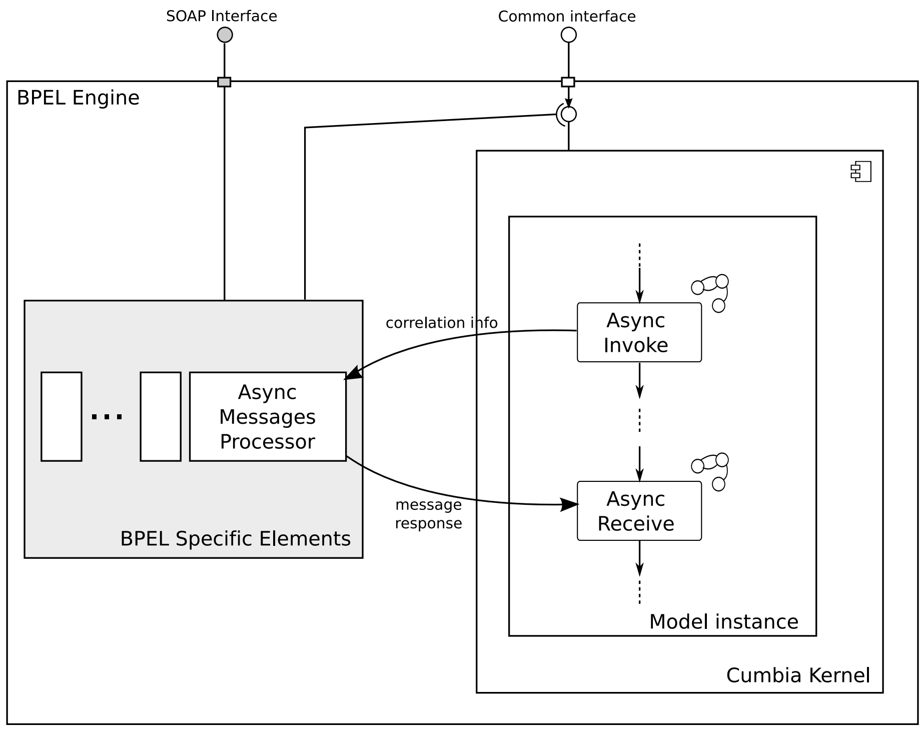


Figure 4.20 Asynchronous messages in a BPEL engine

Figure 4.20 shows an example of this in BPEL. In a BPEL engine implemented in Cumbia[16], asynchronous messages require back and forth interaction between model elements and some other BPEL specific elements located outside of the Cumbia Kernel. When an asynchronous request is done, the corresponding open object (`AsyncInvoke`) contacts the 'Async Messages Processor' and provides *correlation information.* When the response message is received through the SOAP interface, this information is used to locate the adequate instance that sent the outgoing message. Finally, the response message is delivered.

[16]Section 7.7.3 provides more information about the implementation of a BPEL engine on top of Cumbia.

4.6 A sample metamodel: `MiniBPMN`

We have already introduced the main concepts about open objects and about the mechanisms to use and manipulate them. The goal of this and the ensuing sections, is to illustrate the usage of Cumbia and of the open objects to support, execute, and extend a workflow language. This section will thus present the development of the engine for a new workflow language called `MiniBPMN`. The goal of `MiniBPMN` is not to model real-life workflow processes, but it is to illustrate the main characteristics of Cumbia based engines. Thus, the language has very simple structure and semantics. In order to be able to illustrate the structure of `MiniBPMN` processes, a subset of BPMN's notation can be used. Figure 4.21 shows a sample `MiniBPMN` process.

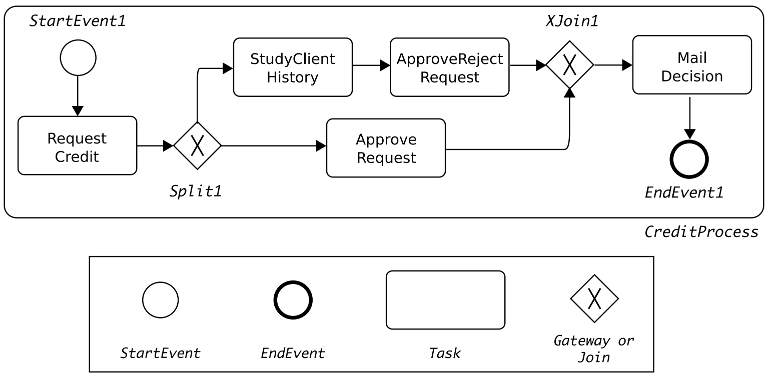


Figure 4.21 `MiniBPMN` sample process with legend

In the following sections we will present the steps necessary to create an engine for this language. These steps starts with the design of the language, its elements, its abstract syntax, and its semantics. Then, a corresponding metamodel has to be designed and implemented. This step includes describing the state machines of the open objects, and writing the code of the entities and the actions. After the metamodel is fully implemented, the engine is completed by implementing additional functionalities not provided by the Cumbia Kernel. Section 4.7 shows how this engine is used and describes the execution of a sample process. Chapter 6 provides in more detail the process to design and implement a workflow engine.

4.6.1 Structure of `MiniBPMN`

The first step to design a metamodel to support a language like `MiniBPMN` is to define its abstract syntax. In particular, this requires identifying the concepts in the language, naming them, and establishing their structure. The result is a metamodel like the one depicted in figure 4.22.

In short, a `Process` in `MiniBPMN` consists of a number of `Tasks` that have to be executed in a specific order. `Events` mark the beginning and the end of the execution: the process starts with a `StartEvent`, and finishes with an `EndEvent`. Each `Process` contains a number of `BasicElements`, which can be `Tasks`, `Gateways`, `Joins`, `Events`, or `(Sub)Processes`. `Gateways` are used to route the control flow depending on conditions: for each condition, there is a `Gate` in the `Gateway`. Since conditions are not necessarily exclusive, it is possible to have concurrent flows in the process. Because of this, two kind of joins (`XORJoin` and `ANDJoin`) are included and they can be used to synchronize parallel flows; their semantics are similar to that of the corresponding elements in BPMN.

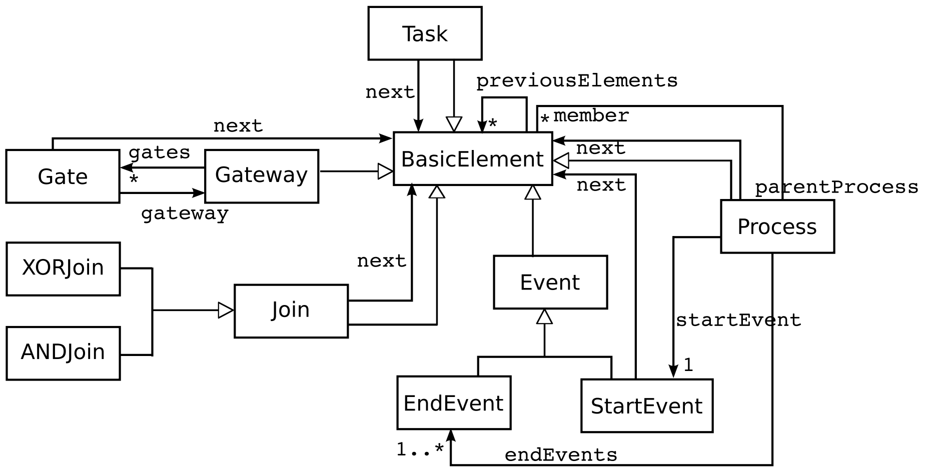


Figure 4.22 Structure of `MiniBPMN`

The execution semantics of `MiniBPMN` is inspired on token passing, although tokens are not reified. In this sense, the semantics of `MiniBPMN` is more similar to the semantics of BPMN, than to the semantics of YAWL. A `Process` starts its execution when the control flow reaches its `StartEvent`, and the execution finishes when *one of its* `EndEvents` receives the control flow. The control flow passes from one `BasicElement` to the next one, and only `Tasks` that currently have the flow can be in execution at any given point. `Tasks` receive the control flow from one element, and always pass it to one element. Instead, `Gateways` receive it from one element, but can pass it to several if several conditions evaluate to true. Finally, `ANDJoins` synchronize the control flow coming from several sources, but pass it just to one other element. The last rule is that when a `Process` finishes its execution, all the elements contained in it must finish their execution as well.

To implement the metamodel, the first step is to write the metamodel specification. This specification defines the elements in the metamodel and the relationships between them (this is the info depicted in figure 4.22). In addition, this file indicates the resources associated with each element: the classes that implement each element's entity, and the structure of their respective state machines. In section 4.6.2 we will present in detail two of the open objects in `MiniBPMN`, namely `Process` and `Task`. The definition of the metamodel and the state machines also requires the specification of roles: in section 4.6.3 we will analyze all the roles used in `MiniBPMN`.

After defining the structure of the metamodel and defining the state machines of each open object, the following step is implementing the entities and the actions: each one of those is implemented in a Java class, using the framework included in the Cumbia platform. Moreover, the definition and implementation of metamodels does not have to be done by hand: a graphical editor for Cumbia metamodels has been developed and it outputs the XML definitions and constructs the skeletons of classes for entities and actions (see chapter 6). After using this editor, the developer of a metamodel only needs to complete the implementation of the methods.

4.6.2 Process and Task

We are now going to analyze two of the main elements of `MiniBPMN` and we will discuss their entities and the structure of their state machine. In the first place, both `Process` and `Task` extend `BasicElement`. Thus, their state machines share the basic structure of the state machine of `BasicElement` (see figure 4.23). A `BasicElement` has two states, *Inactive* and *Active*, and it goes from the former to the latter when either the element before it (`[PE]`, previous element) is deactivated, or when the condition of a `Gate` that targets the element (`[PG]`, previous gate) evaluates to true (section 4.6.3 will present more details about the roles involved). On the other hand, a `BasicElement` is deactivated when the `Process` that contains it (`[PROCESS]`) finishes its execution.

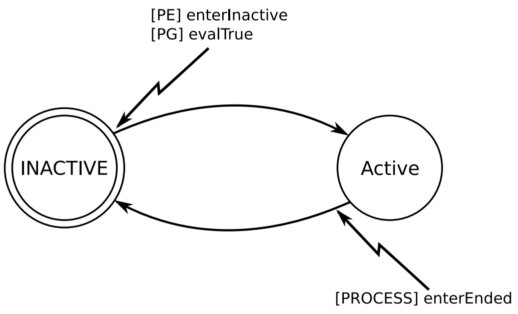


Figure 4.23 State machine of `BasicElement`

A `Process` is not much more complicated than a `BasicElement`: it has a very simple entity, and its state machine only has a few additional things (see figure 4.24). The methods in the entity are used only to activate the process, by giving the control flow to the `StartEvent`, or to abort its execution. The method `activate()` also generates an event called `activate`, which is expected by the state machine. Therefore, the state machine of `Process` takes the transition from *Inactive* to *Active* when said method is invoked. Similarly, the method `abort()` of the entity (`[ME]`) generates the event `aborted`, and when it is invoked the state machine goes from *Active* to *Inactive*. Furthermore, this transition has an action associated that aborts the execution of every element contained in the `Process`.

On the other hand, a `Process` can also finish normally when one of its `EndEvents` receives the control flow. In such cases, the action *abortMembers* is also executed to make sure that every element contained in the `Process` finished its execution.

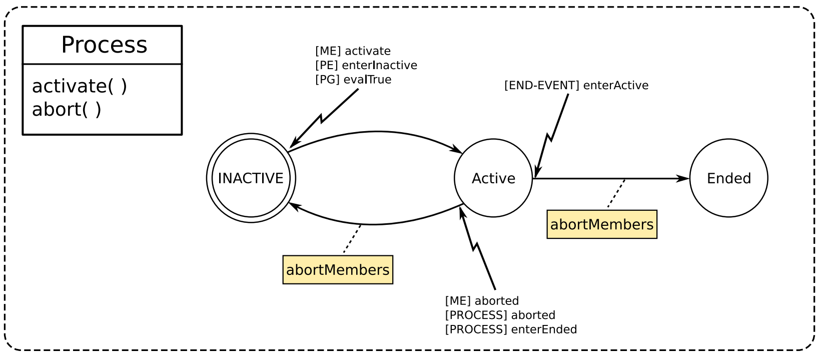


Figure 4.24 Open object for `Process` in `MiniBPMN`

The element called Task is a BasicElement that performs activities in a Process. However, the element Task itself is abstract in the sense that it does not do anything specific. Instead, the element Task should be extended in order to make it perform specific actions. For instance, it could be extended to make it consume web-services, send emails, contact users, transform data in some way, or interact with external applications. Because of this generic nature, the life-cycle of Task is also very generic (see figure 4.25): when it is activated, it gets some initial data, then it uses that data and obtains some results, and then it stores those results.

The entity of Task reflects this life-cycle. The method setData() is invoked by the action *getData* and generates the event dataReceived. The method executeTask() is invoked by the action *processData* and it generates the event dataProcessed when the execution of the task concludes. This is the method that extensions to Task must override in order to have useful Tasks. Finally, the method saveData() is invoked by the action *saveData* and generates the event dataSaved.

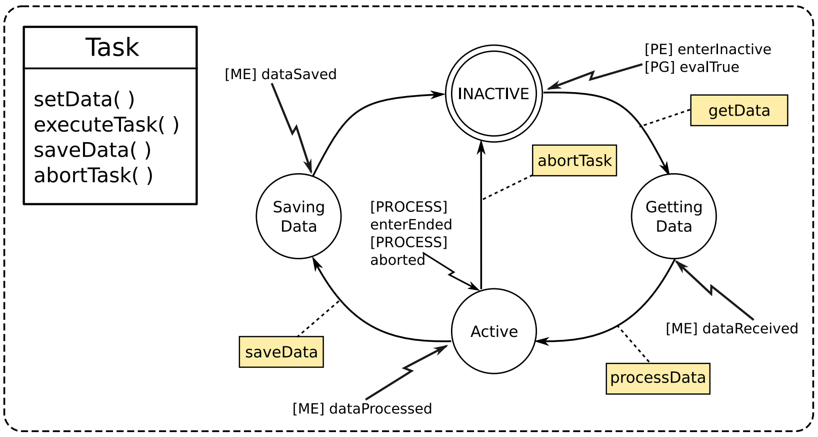


Figure 4.25 Structure of Task in MiniBPMN

4.6.3 Roles in MiniBPMN

The goal of this section is to describe the roles used in MiniBPMN. This will serve to explain the roles mechanism itself, and to illustrate the navigation language.

- [PE]: this role name stands for ‘Previous Element’ and it is fulfilled by any element followed by the base element. This role is described in BasicElement with the following navigation expression:

```
#self.previousElements
```

- [PG]: this role name stands for ‘Previous Gate’ and it is fulfilled by a Gate (of a Gateway) that targets the base element. This role is described in BasicElement with the following navigation expression:

```
#self.parentProcess.member.
    {? #this isOfType "Gateway"}.gates.
    {?  #this.next == #self }}
```

This expression can be decomposed in the following way:

- `#self` : is the base element.
- `parentProcess` : is the `Process` that contains the base element.
- `member` : is any `BaseElement` that is a member of the `Process`.
- `{? #this isOfType "Gateway"}` : filters the members to keep only those of type `Gateway`.
- `gates` : is any `Gate` of a `Gateway` previously identified.
- `{? #this.next == #self}` : filters the gates to keep only those that target the base element.

An alternative expression to describe the same role is the following:

```
#self.previousElements.
  {? #this isOfType "Gateway"}.gates.
  {?  #this.next == #self }
```

- `[PROCESS]`: this role is used to describe the `Process` that contains the base element. This role is described in `BasicElement` with the following navigation expression:

```
#self.parentProcess
```

- `[END-EVENT]`: this role is used to describe the end events of the base element. This role is only defined for the type `Process`. The navigation expression to describe the role is:

```
#self.endEvents
```

- `[ME]`: this role is used for events generated by the entity of the element that owns the state machine. The role `[ME]` is predefined for every metamodel, and it is equivalent to the navigation expression:

```
#self
```

To illustrate the meaning of these roles, we are going to evaluate some of them in the sample process depicted in figure 4.21. Table 4.1 shows the results of this evaluation, using different base elements for each role. Note that some of these evaluations result in an empty set of elements.

4.6.4 An engine for MiniBPMN

Besides the definition and implementation of the `MiniBPMN` metamodel, there are not many other things required to have a functional engine for this language. The Cumbia Kernel and the metamodel, already provide an implementation for many functionalities. In the first place, the Cumbia Kernel loads `MiniBPMN` models expressed using the Cumbia XML schema. Thus, it is not necessary to implement any additional parser. Furthermore, the kernel is responsible for instantiating the models when new cases are going to be enacted, and it includes a generic persistence system to store and maintain the state of the execution.

Nevertheless, there are a few elements specific to `MiniBPMN` that must be implemented (see figure 4.26). For instance, if `Tasks` in a `MiniBPMN` process need to interact with external applications, then the communication mechanisms can be implemented as part of the metamodel, or they can be part of the engine.

Table 4.1 Some roles evaluated in the sample process

Role	**Base Element**	**Matching Elements**
PE	XJoin1	{ApproveRejectRequest, ApproveRequest}
	ApproveRequest	{}
	RequestCredit	{StartEvent1}
PG	StudyClientHistory	{GateStudy}
	ApproveRequest	{GateApprove}
	RequestCredit	{}
PROCESS	MailDecision	{CreditProcess}
	RequestCredit	{CreditProcess}
END-EVENT	CreditProcess	{EndEvent1}
	MailDecision	{}
ME	CreditProcess	{CreditProcess}
	MailDecision	{MailDecision}

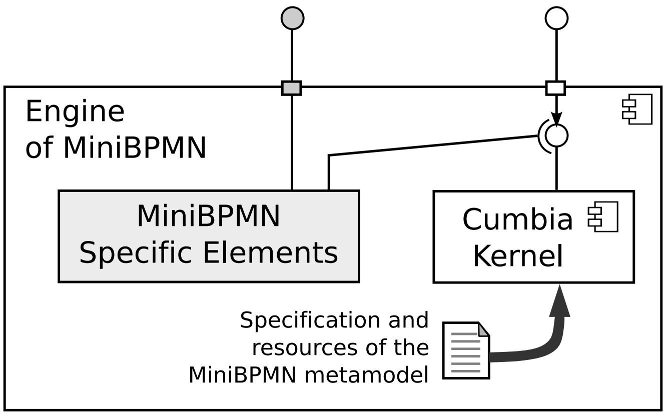


Figure 4.26 Architecture of the engine for `MiniBPMN`

4.7 A sample `MiniBPMN` model

This section illustrates the usage of open objects and of `MiniBPMN` with a sample workflow. This process was taken from the banking context and it is used to grant "fast credits". In this chapter, we only consider the *control flow* concern of this process; however, the complete workflow also includes other concerns like time and resources, which will be discussed in chapter 5.

The process modeled starts when a client issues a credit request. If the amount requested is less than 1000 USD, the credit is approved. Otherwise, the financial history of the client is studied, and then the results of those studies are used to decide on the approval or rejection of the credit. Finally, the client that requested the credit is informed by email about the result of the process.

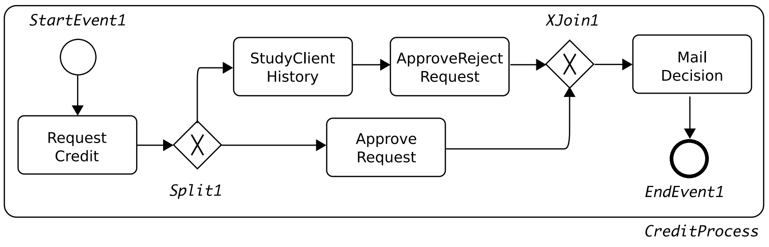


Figure 4.27 `MiniBPMN` sample process

Figure 4.27 presents a graphical representation of the `MiniBPMN` model that represents the process that was described above. Figure 4.28 shows the same model, but this time it is represented using an UML object diagram, where each object corresponds to an open object.

The goal of this diagram is to show which open objects are used in the sample process, and show the identifiers of the instances employed. For example, the sample model includes two instances of the `MiniBPMN` element called `Gate`, and these instances are identified as *gateStudy* and *gateApprove*. Furthermore, it should be clear that the instances in this diagram are not 'executable instances': they are part of a model definition, they have a type, and they have a name; however, they do not have an execution state, because a model definition is not run. In a later stage, this definition and all its open objects have to be instantiated to be run inside the Cumbia Kernel. Only at that point it makes sense to talk about the state of the open objects.

For clarity, the diagram only shows the entities of the open objects. Additionally, it also shows most of the connections declared in the model definition between these entities. We only removed from this diagram the relations 'member' and 'parentProcess' that exist between the `Process` and all the other elements (but the `Gates`). This diagram can also be used to validate the evaluation of the roles presented in section 4.6.3 (cf. table 4.1).

A final note about this sample model is that all the instances of `Task` are really instances of open objects that extend the type `Task`. These extensions are all different and they specialize the method `executeTask( )` in order to have a real, specific behavior in the process.

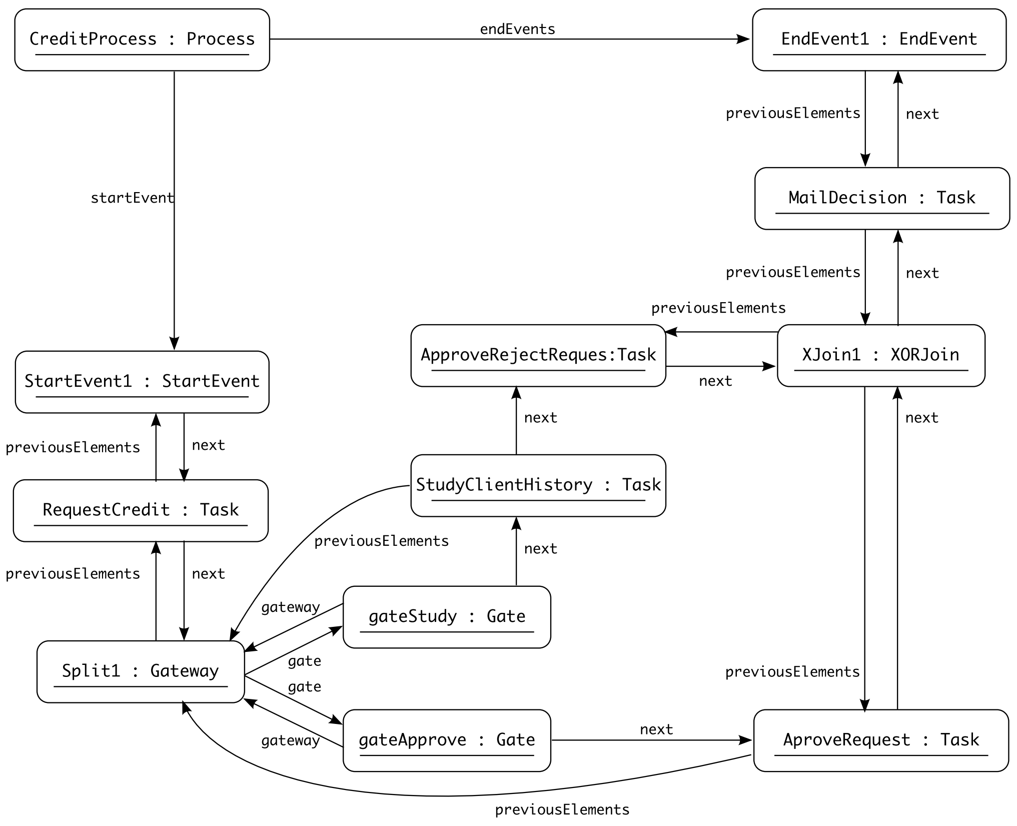


Figure 4.28 Objects diagram corresponding to the sample process

4.8 Extensions to the `MiniBPMN`

This section shows how a language can be extended with new concepts and constructs. In particular, we are going to extend `MiniBPMN` in four different ways, using the previously discussed extension mechanisms.

The first two extensions will introduce a new concept known in BPMN as *Ad Hoc Subprocesses* [Obj08]. In those, tasks are not structured, but *all of them* have to be executed in a *sequential* order defined at run time. In the revised list of control-flow patterns [RtHvdAM06], this is known as Interleaved Routing (pattern number 40). To achieve this goal, we will introduce two new constructs into `MiniBPMN`: Extension 1 is an extended kind of `BasicElement` called `AdHocBlock`, which will function as BPMN's Ad Hoc Subprocess; Extension 2, is an extension to `Task` called `AdHocTask`; an `AdHocBlock` can only contain `AdHocTasks`. The other two extension scenarios are unrelated: Extension 3 introduces a minor extension to element `Process`, and Extension 4 introduces a new element called `ExecutionLogger`.

Figure 4.29 shows the final structure of `MiniBPMN` after the application of all the extensions. In this figure the new elements have been shaded, and a bold arrow has been used to represent an extension relation that is different from other extension relations in the metamodel. In the following sections we provide all the details about this.

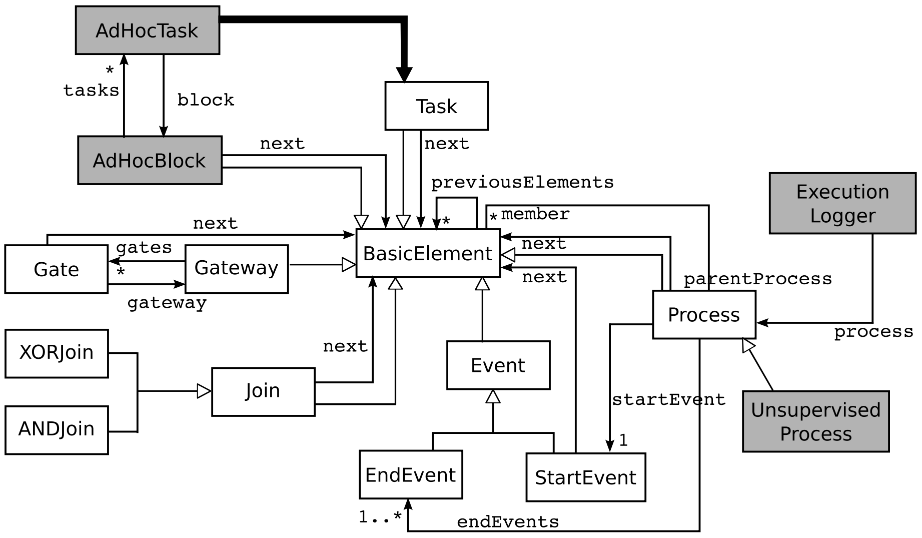


Figure 4.29 Extended MiniBPMN metamodel

4.8.1 Extension 1: AdHocBlock

Figure 4.30 summarizes the extensions to the metamodel that are necessary to introduce *Ad Hoc Subprocesses* in MiniBPMN. This section will focus on the new element AdHocBlock, while the next section will focus on AdHocTask. Furthermore, it must be noted that in the figure the extension relationship between AdHocTask and Task, which is described in section 4.8.2, has been depicted with a wide arrow to highlight that it is different from the relation between AdHocBlock and BasicElement.

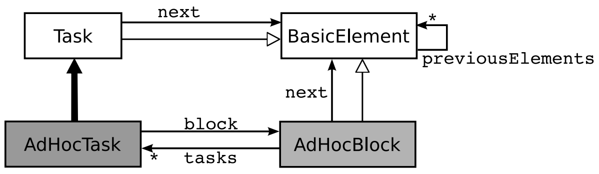


Figure 4.30 Extended MiniBPMN metamodel

In order to introduce an AdHocBlock into MiniBPMN, we added a new element that extended BasicElement. In this case, the extension consisted in providing a new interface and a new implementation for the entity, and also a new state machine. However, to minimize compatibility problems, we complied with the following restrictions:

1. The interface of AdHocBlock *extends* the interface of BasicElement.

2. The class that implements the entity of AdHocBlock *extends* the class that implements the entity of BasicElement.

3. The state machine of AdHocBlock is *compatible* with the state machine of BasicElement. In this context, this means that the state machine of AdHocBlock can be constructed by applying extensions to the state machine of BasicElement.

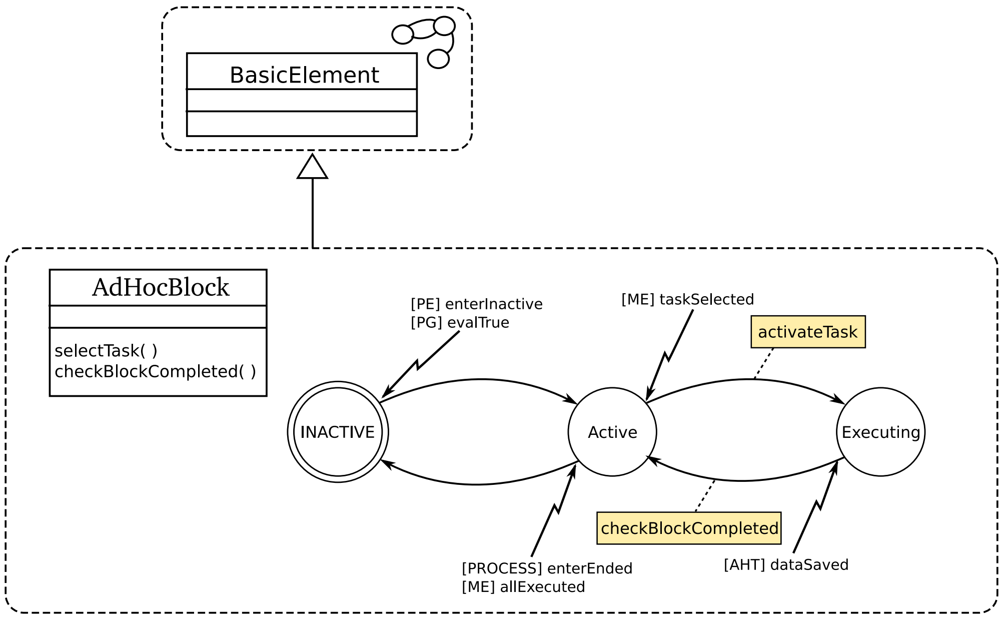


Figure 4.31 Extended element: `AdHocBlock`

Figure 4.31 shows the structure of the new element. Listing 4.27 is the extract from the extended metamodel specification where the new element is described. First of all, line 2 specifies the state machine to use with this new element. Then (lines 4–9), it describes the new entity, by specifying its interface and class, and also the name of the events that it can generate. Afterwards (lines 12–13) the code specifies the relations of the new element. Finally (lines 15–17) it defines the new role `[AHT]` which the state machine requires: this description specifies that [AHT] have to be of type `AdHocTask`, and also provides a navigation expression (`#self.tasks`) to find those elements.

Listing 4.27: Definition of the `AdHocBlock` extension (without the state machine)

```
1  <extended_type name="AdHocBlock" extends="BasicElement">
2    <new_state_machine name="AdHocBlockSM" />
3
4    <!-- Entity -->
5    <new_entity entityClass="cumbia.minibpmn.AdHocBlock"
6                interface="cumbia.minibpmn.IAdHocBlock">
7      <new_event name="taskSelected" />
8      <new_event name="allExecuted" />
9    </new_entity>
10
11   <!-- Relations and Roles-->
12   <new_relation name="next" cardinality="simple"
13                 targetTypeName="BasicElement"/>
14   <new_relation name="tasks" cardinality="multiple"
15                 relationType="sequence" targetTypeName="AdHocTask"/>
16
17   <new_role name="AHT" description="Ad Hoc Tasks within the block">
18     <role_detail type="AdHocTask">#self.tasks</role-detail>
19   </new_role>
20 </extended_type>
```

4.8.2 Extension 2: AdHocTask

The second element added is an extension to `Task` that is called `AdHocTask`. `AdHocTask` introduces a number of changes with respect to `Task`. In the first place, the state machine is extended with the addition of a transition, and of an intermediate state. From the perspective of the entity, `AdHocTask` extends the interface and its implementation by adding the method `selectTask( )`. Finally, an additional event is declared. Figure 4.32 shows the new element.

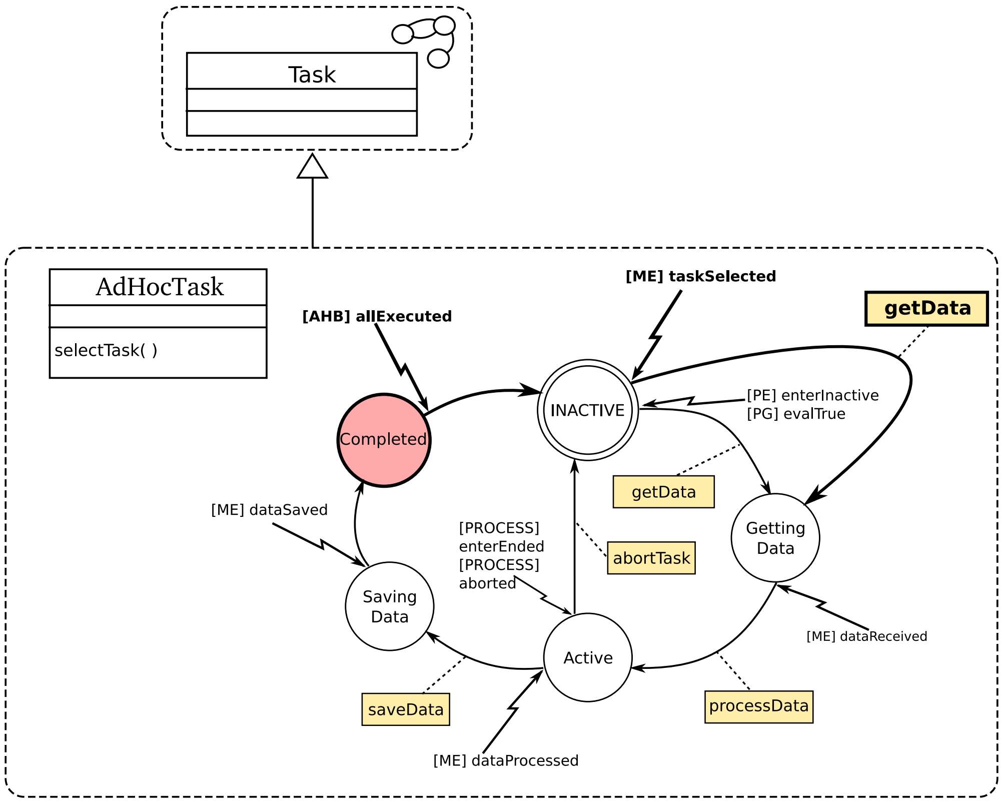


Figure 4.32 Extended element: `AdHocTask`

Listing 4.28 shows the extract of the metamodel extension that describes the new element `AdHocTask`. Lines 4 to 7 describe the extended entity, including the new event that can be generated. Then lines 10 to 35 specify the extensions to the state machine: first, an intermediate state is added between states *SavingData* and *Inactive* (lines 12–22); afterwards, an additional transition between *Inactive* and *GettingData* is added (lines 25–35). The last part of the code defines a new relation called *block*, and the new role `[AHB]` that serves to refer to the `AdHocBlock` that contains the `AdHocTask`. Note that this role is used in the transition *allTasksExecuted* that is added with the intermediate state.

4.8.3 Extension 3: Unsupervised Process

For the third extension to `MiniBPMN` we are going to consider a much simpler requirement. In this extension scenario, the new requirements is to have unsupervised processes that send an email to the administrator when they are completed. This extension can be implemented by defining an extension to `Process`, and adding a new action to its state machine.

Listing 4.28: Definition of the `AdHocTask` extension

```
<extended_type name="AdHocTask" extends="Task">

  <!-- Extensions to the entity of Task -->
  <new_entity entityClass="cumbia.minibpmn.AdHocTask"
              interface="cumbia.minibpmn.IAdHocTask">
    <new_event name="taskSelected" />
  </new_entity>

  <!-- Extensions to the state machine of Task -->
  <state_machine_extensions>
    <!-- Add Intermediate State -->
    <add_intermediate_state transitionName="deactivating"
                            location="after">
      <additional_state name="Completed" enter_event="enterCompleted"
                        exit_event="exitCompleted">
        <additional_transition name="allTasksExecuted">
          <source_event source_name="AHB" event_name="allExecuted" />
          <before_event name="beforeAllTasksExecuted" />
          <after_event  name="afterAllTasksExecuted" />
        </additional_transition>
      </additional_state>
    </add_intermediate_state>

    <!-- Add Transition -->
    <add_transition name="taskSelectedTransition"
                    source_state="Inactive" successor="GettingData">
      <source_event source_name="ME" event_name="taskSelected" />
      <before_event name="beforeTaskSelected" />
      <after_event  name="afterTaskSelected" />
        <actions>
          <action name="getData"
                  class="cumbia.minibpmn.task.GetData" />
      </actions>
    </add_transition>
  </state_machine_extensions>

  <!-- Relations and Roles-->
  <new_relation name="block" cardinality="simple"
                targetTypeName="AdHocBlock"/>

  <new_role name="AHB" description="AdHocBlock that contains the task">
    <role_detail type="AdHocBlock">#self.block</role_detail>
  </new_role>
</extended_type>
```

Listing 4.29 shows the definition of this new extension. Listing 4.30 shows the structure of the code of the new action. As it can be seen, integrating the action with the state machine does not require any complex code.

Listing 4.29: Definition of the UnsupervisedProcess extension

```
<extended_type name="UnsupervisedProcess" extends="Process">
  <state_machine_extensions>
    <!-- Add additional action to the transition -->
    <add_actions transitionName="processEnded">
      <action name="notifyProcessEnded"
              class="cumbia.minibpmn.extension.NotifyProcessEnded" />
    </add_actions>
  </state_machine_extensions>
</extended_type>
```

Listing 4.30: Implementation of the new action NotifyProcessEnded

```
public class NotifyProcessEnded implements IAction
{
  public void execute( EventNotification event, Transition transition,
      IOpenObject element )
  {
    IProcess p = (IProcess) element;
    // Send the email with the notification
    ..
  }
}
```

4.8.4 Extension 4: Execution Logger

The fourth extension scenario adds an element to MiniBPMN that does not extend from any existing elements. This new element is called ExecutionLogger, and its responsibility is to create a log about the execution of all the Tasks of a Process. Although none of the existing elements will know about the new one, ExecutionLogger will have relations to other elements and will react to events produced by them.

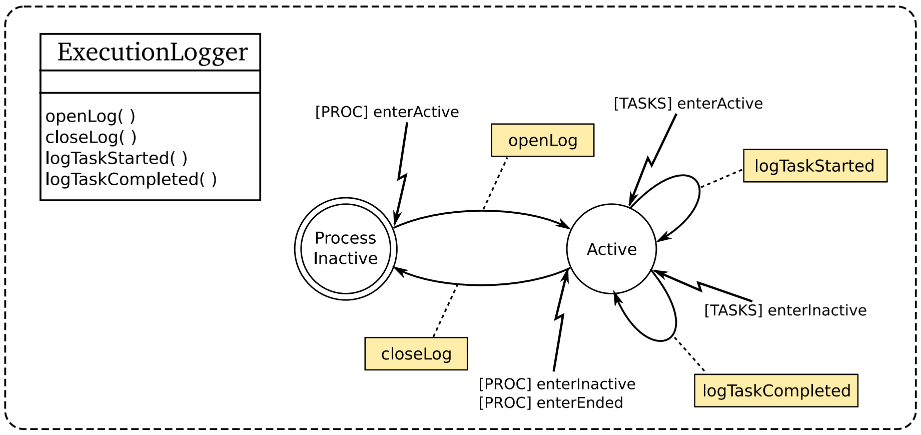


Figure 4.33 New element: ExecutionLogger

Figure 4.33 shows the structure and the state machine of this new element, and shows how it is related to `Process`. Listing 4.31 shows how this new element is declared as a new element within the extension to `MiniBPMN`.

Listing 4.31: Definition of the `ExecutionLogger` extension

```xml
<metamodel name="MiniBPMN-Extension" extends="MiniBPMN" version="0.1">

  <state_machine_reference name="ExecutionLoggerSM"
                           file="executionLoggerXM.xml" />

  <type name="ExecutionLogger"
        entityClass="cumbia.minibpmn.extension.ExecutionLogger"
        interface="cumbia.minibpmn.extension.IExecutionLogger"
        statemachine="ExecutionLoggerSM">

    <!-- Relation -->
    <relation name="process" cardinality="simple"
              targetTypeName="Process"/>

    <!-- Roles Definition -->
    <role name="PROC" description="Monitored Process">
      <role_detail type="Process">#self.process</role_detail>
    </role>
    <role name="TASKS" description="Monitored Tasks">
      <role_detail type="Task">#self.process.member</role_detail>
    </role>
  </type>
</metamodel>
```

4.9 Summary

This chapter has presented the core elements of our proposal. It started with a discussion about metamodeling platforms, and then it introduced the notion of open open objects as the main modeling abstraction for our platform. Open objects were thoroughly presented in this chapter and concrete examples were used to illustrate their characteristics. Afterwards, we showed how open objects are assembled together to form metamodels, how metamodels are used to create models, and how engines are built to execute those models. The last part of the chapter presented a detailed example of the usage of the platform, based on an ad hoc workflow specification language called `MiniBPMN`.

In chapter 3 we briefly introduced some characteristics of the open objects that are essential to the construction of workflow engines. We now revisit these characteristics.

- **Various execution models supported.** In this chapter we have shown that the execution model of the open objects is not related to any particular workflow language. In fact, it is not at all related to workflows. Therefore, in principle it should be possible to implement several workflow execution models on top of these elements. In chapter 7 we present several case studies that confirm this idea.

- **Externalized state and externalized interaction.** In this chapter we presented in detail the methods to publish the run time state of open objects, and to use this information to coordinate the execution of elements.

In this chapter the coordination and interaction was limited to elements in the same model, but in chapter 5 we show how models conformant to different metamodels can interact using the same mechanisms.

- **Compatibility with the kernel, and a base development framework for developers.** Throughout the examples presented in this chapter, we have shown that the open objects and the Cumbia Kernel provide a number of features and functionalities that are often required when workflow applications are developed. Therefore, developers of new languages do not have to focus on these features, and can focus on characteristics that are specific to the language and its elements.

- **Explicit extension mechanisms.** The extension mechanisms that open objects provide were extensively presented in this chapter. These mechanisms allow both for horizontal extensions (extensions to the functionalities) and for vertical extensions (specialize the implementation of elements) [EVLA+03]. The ideas about multiple concern specific languages, which are presented in chapter 5, are another way of introducing horizontal extensions. This chapter also presented a full example where extensions to the language were obtained by applying the aforementioned extension mechanisms to one metamodel.

- **Decoupled interfaces.** In this chapter we mentioned and illustrated the specific interfaces implemented by the entity of each open object. Moreover, we also presented the basic characteristics of the general interfaces that are shared by every open object and serve to interact with it. These interfaces, contribute to maintaining low the coupling between elements in a metamodel.

The model presented, and in particular its behavior, has a number of elements that elevate its complexity. However, these elements were included with the explicit goal of augmenting the expressiveness of the model, and enabling the representation of more languages. One example of this are the two mechanisms of interaction available: one is synchronous and based on actions and invocations to entities' methods; the other one is asynchronous and based on events. When the model was designed, we faced the alternative of selecting only one interaction mechanism, but we considered a problem the enormous reduction in expressiveness. In chapter 7 we discuss a case study where we used only the asynchronous interaction mechanisms to implement a Petri net interpreter. In spite of the relative simplicity of Petri nets, the metamodel that we had to implement turned out extremely complex because of the lack of synchronous mechanisms of interaction.

Another decision that increased the complexity of the model, but also increased its expressiveness, has to do with the events generated when a transition is triggered. In the case studies developed, we have frequently seen that the number of events available largely exceeds the number of events required: generating two events per state change (or even one) is enough to achieve the coordination required in most metamodels. However, having the four events shown in figure 4.4, facilitates the design of the state machines because more information is available about each event.

The expressiveness of the open objects model was also a main concern when we decided against restricting what can be included in the methods of entities, or in the actions associated to transitions. This decision also elevates the complexity of the model and, by making these implementations totally opaque, renders difficult activities such as the analysis and verification of the metamodels and the models.

The information presented in this chapter is complemented by chapter 5, which focuses on using open objects to support multiple concern specific workflow languages. Chapter 6 complements what these two chapters presents by showing how users should use the elements and tools that Cumbia provide to build workflow engines for new languages. Finally, chapter 7 illustrates with concrete scenarios and case studies, all the elements that this chapter presented.

5 Coordination of Multiple Concern Specific Models

Most workflow engines describe in a single artifact all the relevant details of a workflow. These artifacts, which may be diagrams, textual programs, or xml representation of models, are usually described with languages that include a large number of elements and handle general concepts. This has led to two problems. On the one hand, the artifacts in which workflows are described tend to become big and difficult to manage as workflows become more complex. On the other hand, the artifacts are difficult to understand and maintain because the languages are not very suitable or do not manage proper domain concepts. To solve these problems, we propose to use multiple Concern Specific Workflow Languages (CSWLs). These languages offer various advantages compared to using a single language. First of all, they can be more suitable because they use concepts that are very close to each concern. Since they focus on a small part of the whole workflow, this also makes each one of them simpler when compared to generic workflow languages. As a result they are also easier to implement, to maintain, and to change when necessary. A downside of using CSWLs is that one often needs to use multiple ones to specify a full workflow application. As a result one has to manage the integration of several languages, and this raises the overall complexity of the systems. However, this complexity can be mitigated by providing adequate mechanisms to relate the languages and also to relate the models built with those.

In the previous chapter we showed how to implement engines for new workflow languages using the Cumbia platform as their base. Nevertheless, we only considered the case where a single language was used to describe all the relevant details of a workflow. Conversely, in this chapter we show how Cumbia and the open objects support the integration and interaction of multiple concern specific worklfow languages. In particular, this chapter focuses on the mechanisms to coordinate the execution of concern specific models.

This chapter begins by briefly presenting some existing approaches for model composition, together with a framework to compare such strategies and tools. Afterwards, we present our strategy to use multiple concern specific workflow languages. This strategy is based on explicitly expressing how to coordinate concern specific models, and using the interaction mechanisms offered by open objects to achieve this coordination. This strategy is based on two additional elements of the Cumbia platform. The first one is CCL, the Cumbia Composition Language, which is presented in section 5.2. The second one is the Cumbia Weaver, which we present in section 5.3. CCL is the language to describe how to coordinate the execution of concern specific models. CCL is also independent of the concern specific languages, and thus it can be used to describe the coordination between models even if they are specified with different metamodels. On the other hand, the Cumbia Weaver is the element in the architecture of Cumbia that serves to integrate different engines and to interpret CCL programs. The chapter is concluded with a complete example that illustrates the usage of concern specific workflow languages, and their coordination using CCL and the Cumbia Weaver.

5.1 Model composition and coordination

In the previous chapters we showed how to define workflow languages using metamodels and how to build models using those metamodels. When concern specific workflow languages are used, several metamodels are used (one for each CSWL), and several models (one for each concern specification) have to be related to capture the complete semantics of the workflow. In Cumbia this is achieved by having various concern-specific models and coordinating their execution. The coordination mechanisms used, which can be considered a model composition technique, are introduced in this section.

This section is organized in two parts. First, the general idea of model composition is introduced and the framework introduced by Jeanneret et al. for comparing model composition techniques is presented [JFB08]. The second part of the section presents the general ideas of the model composition strategy used in Cumbia.

5.1.1 Model composition techniques

To avoid the problem of having every detail of a system specified in one huge model, in many domains it is common to use multiple complementary models. Since each of these models only captures the properties relevant for a specific viewpoint of the system, these models are significantly simpler and easier to manage. Furthermore, the modeling language can vary across different viewpoints [JFB08] so that suitable elements and constructs are available.

Different types of relations can be established between viewpoint-specific models. For instance, a common type is that of equivalence relations between elements of different models: these relations indicate which model elements represent the same system elements. The kind of relations that are most relevant to our approach are those that specify how elements from different models should interact when workflows are enacted.

Establishing relations between elements in different models is not trivial, and it has been shown that human intervention is frequently required [FBJ$^+$05]. Rules and heuristics to automatically discover relations have been studied (e.g. based on naming conventions), but in it is often the case that they cannot find all the meaningful relations between two models. Moreover, establishing these relations is not necessarily easy for humans and it requires a good knowledge about the models that are composed.

In the past, several model composition strategies and tools have been proposed. The characteristics of those strategies have been determined by factors such as the type of models composed, the available technologies, and the intention of the composition. These factors have to be taken into account when selecting a composition strategy in a specific modeling context.

Two representative and well known model composition approaches that present very different characteristics are AMW [FBJ^{+}05] and the Motorola WEAVR [CvdBE05, CvdBE07]. The first one is a model weaving tool based on ATL transformations, which depends on an explicit weaving metamodel. With this metamodel it is possible to describe a weaving model that specifies how some source models have to be composed to obtain a single model. The actual composition is performed by a weaver that applies ATL transformations derived from the weaving model information. A further characteristic of AMW is its focus on the structural aspect of the systems and of the composed models.

On the other hand, there is WEAVR, an aspect-based model weaver. WEAVR was developed at Motorola to coordinate and to reason about the coordination of crosscutting concerns described with separate models. Therefore, WEAVR focuses on the interaction between the elements in those models, rather than focusing on other kind of relations between them. Since WEAVR is an aspect-based approach, they adopted concepts from AOP such as joinpoints, aspects and advices.

AMW and WEAVR share some characteristics, such as the usage of code generation. They differ in others, such as the emphasis in structural composition in AMW versus the emphasis on coordination in WEAVR. To select one of them as the composition strategy or tool for a certain project, they have to be analyzed from several points of view. To guide this selection, a reference process framework for model composition was proposed in [JFB08]. This process identifies several aspects of the model composition strategies that are worth comparing. The following are these aspects, which are illustrated with the analysis of AMW proposed by the authors of the framework. In section 5.5 we will use this framework to analyze our own approach.

- **Ordering**

  Since most compositions require more than one step, one relevant aspect is the order in which those composition steps are executed.

  E.g.: In AMW the composition steps are executed according to the order of application of the ATL rules.

- **Ordering Flexibility**

  This aspect of the framework refers to the flexibility in the ordering of the composition steps, which in some cases is not rigidly fixed by the user who describes the composition.

  E.g.: In AMW the ordering is inferred by the ATL engine.

- **Stopping Criteria**

  This aspect identifies the criteria to determine that the composition procedure has finished.

E.g.: In AMW the composition process finishes when no more ATL rules apply.

- **Stopping Criteria Flexibility**

  This aspect refers to the flexibility of the criteria to stop the composition procedure. In some strategies , the composition has a fixed number of steps, while in others the number of composition iterations are difficult to predict and depend on the stopping criteria.

  E.g.: Since AMW depends on ATL rules, the flexibility of the stopping criteria depends on the ATL engine.

- **Contribution**

  In this framework, the term *contribution* refers to the elements that are composed.

  E.g.: In AMW, a contribution is a weaving model.

- **Contribution Granularity**

  This aspect refers to the granularity of the descriptions of elements which are composed (the contributions).

  E.g.: In AMW, the granularity of the contribution are the elements in a weaving model.

- **Contribution Selectivity**

  This aspect refers to the kind of elements that can be selected to be a contribution, and the limitations on this selection.

  E.g.: In AMW the selection is arbitrary and any element in a weaving model can be selected.

- **Location Description**

  In this framework, the term *location* refers to the point in the target model (the composed model) where a contribution is going to be attached. This aspect refers to the mechanism to identify the locations.

  E.g.: In AMW the location for the composition is determined in the ATL transformation.

- **Location Freedom**

  The locations available to do a composition can also be limited. This aspect refers to the kinds of valid locations, and to the mechanisms to describe such locations.

  E.g.: In AMW, locations are described using ATL transformations, and the valid locations are all the locations that can be described using ATL.

- **Combination Description**

  The way in which contributions are *combined* in the composed model varies among strategies and is an important criteria to compare them. The Combination Description aspect refers to the mechanisms to describe the combination strategies employed by each composition strategy.

  E.g.: In AMW the combination depends on both the weaving metamodel and the weaving model.

- **Combination Expressiveness**

  This aspect refers to the composition operators available to do the combination. The expressiveness of the composition strategy depends on the power of these operators.

  E.g.: In AMW the combination expressiveness depends on user-defined operators.

- **Translation**

  Some model composition strategies require source models to be conformant to the same metamodel and thus they apply some translation mechanism. This aspect refers to the technique to perform this translation, when necessary.

  E.g.: AMW employes ATL transformations to translate the models.

- **Fastening**

  This aspect refers to the technique employed to actually *assemble* the composed model.

  E.g.: AMW employes ATL transformations to obtain the composed models.

In addition to the criteria that we just described, we have added an additional one to the framework. This additional criteria is the following.

- **Symmetry**

  This aspect analyzes whether the composition is symmetric or asymmetric. When composition is asymmetric, the source models necessarily have different roles. For example, one of the source models can be the base for the composition and the contributions can be attached to it.

  E.g.: AMW uses a symmetric composition approach. Different models can have different roles, but that is up to the composition operators selected by the user.

5.1.2 Cumbia model composition strategy

The composition strategy that we use in Cumbia is motivated by the need to have several models representing different concerns of a workflow, and by the need to enact that workflow. Especially because of this enactment requirement, the focus of the composition strategy presented in this dissertation is on the coordination of the models. That was our starting point, and we have left for future work the missing research about the means to describe composition at the metamodel level. Conversely, other approaches for model composition have focused on the description of possible compositions at the metamodel level. For example, in the Melusine environment composite domains are created to relate existing domains, and composition models conformant to the composite domains are then used to describe the relations between domain models [EIV05]. The interpretation of the whole applications is distributed among domain specific virtual machines, and a virtual machine for the composition, which references the other ones. Finally, the synchronization between all of these machines is achieved through a mechanism based on AOP.

Another important characteristic of the Cumbia composition approach is that it does not require coordination code and executable code to be generated. On the contrary, Cumbia keeps the models largely independent, and uses the interaction mechanisms offered by the open objects to synchronize the execution of the models.

The composition mechanism employed in Cumbia is *external*. This means that the composition of the models is defined outside of the models themselves, in a separate artifact, *after* the models are created. Because of this, the composition has minimum impact on the models, and this allows models to be reused more easily [EIV06]. This aspect of the strategy is similar to other approaches, such as Melusine [EIV05].

Furthermore, the composition mechanism is *lightweight*. In this context, this means that the relationships established between elements in different models are easy to set up and do not have a profound impact on the models. Similarly, these relationships can be easily removed, leaving the models unaltered. A further consequence of this is that concern specific models are not joined together but remain separate artifacts. Thus, at all times it is possible to identify the parts that make a complete workflow.

There are two elements that are central to the composition strategy in Cumbia. The first one is CCL, a language to establish the composition between models by using the interaction mechanisms offered by open objects. In section 5.2 we will present in detail this language. The second element in this composition strategy is a component of the architecture that we have called the Cumbia Weaver. This weaver plays a central role because it interprets CCL instructions and establishes the relationships between models in accordance to those instructions. Since the relationships are materialized between model instances when their elements are *fastened*[1], the Cumbia Weaver also has some responsibilities related to the creation of the model instances. Finally, the Cumbia Weaver maintains the information about the relationships that it created. Section 5.3.2 presents more information about the Cumbia Weaver and about the overall architecture of Cumbia based workflow engines that support concern specific workflow languages.

A strategy similar to the one presented in this dissertation was previously employed in APEL, a process support system that supports multiple domains that complement and extend a process kernel [DEA98, EVLA$^+$03]. In APEL, concepts belonging to different domains are structured in different metamodels. When it becomes necessary to execute models conformant to those metamodels, instances of those models are linked, at runtime, by a *federation engine*. This linkage is done by means of an AOP machine, which inserts the necessary code to intercept method calls and invoke the code that does the binding between model elements.

5.1.3 A note about the graphical syntax

In order to describe graphically the relations between metamodels, we have developed a small graphical syntax. Using this syntax it is possible to specify for a given situation how the metamodels involved are related. Figure 5.1 shows an example of the usage of this syntax. The meaning of the lines in the syntax is

[1]Cf. the framework of Jeanneret et al. [JFB08].

described in table 5.1.

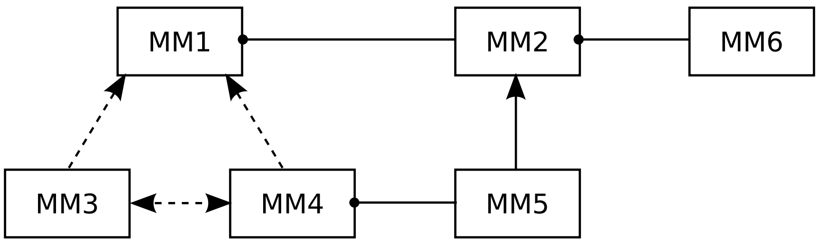
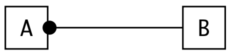
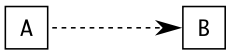
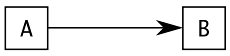


Figure 5.1 Graphical syntax example

Table 5.1 Types of lines in the proposed graphical syntax

	This line indicates that metamodel B is an extension to the base metamodel A. This means that the elements in the extended metamodel can be defined in the base one, can be extensions to elements in the base one, or can be entirely new elements.
	This line indicates that models conforming to metamodel A can be composed to the models conforming to metamodel B. In particular, this means that the former models are prepared to observe and react to changes in the latter models. These reactions can have effects on the models conformant to metamodel B.
	This line indicates a relationship similar to the previous one, but with stronger conditions. The additional requirement is that metamodel A has specific dependencies towards metamodel B. Therefore, models conforming to the former metamodel can be composed only to models conforming to the latter metamodel or to extensions of it.

Based on this description, the semantics of figure 5.1 is the following:

- MM2 is a metamodel that extends metamodel MM1.
- Models conforming to MM3 can be composed with models conforming to MM1.
- Models conforming to MM4 can be composed with models conforming to MM1.
- Models conforming to MM3 can be composed with models conforming to MM4, and vice-versa.
- MM5 is a metamodel that extends metamodel MM4.
- MM6 is a metamodel that extends metamodel MM2.

- Models conforming to MM5 have to be composed with models conforming to MM2 or to MM6.

In this chapter we use this graphical notation in section 5.4 to describe the organization of the metamodels in the sample workflow.

5.2 CCL – The Cumbia Composition Language

CCL is the language that Cumbia uses to describe the composition of models based on Open Objects. Since this composition is based on coordinating the execution of the models, CCL can be considered a coordination language akin to control-driven coordination languages such as Manifold [PA98, AHS93]: it has a role on gluing active pieces of programs (i.e. open objects in different models) to make them behave as an ensemble. Moreover, as the rest of the current section shows, CCL is closer to coordination models and languages that deal with configuration and architectural description, such as Conic [PA98, KMF90], and it is farther to data-driven coordination models and languages, such as Linda.

The central goal of CCL is to offer instructions to link model elements and enable their interaction. The instructions offered are closely related to the interaction mechanisms available in open objects and rely on events, actions, and the invocation of methods in the entities. Because of this, we consider CCL a low level language. To use it, it is necessary to have a good knowledge about the open objects' interaction mechanisms, and about the entities and state machines of the specific elements that must interact. The benefit taken from this low level nature, is that CCL is decoupled from specific metamodels. Therefore, it serves to describe the coordination between any group of Cumbia models, regardless of the languages used to describe each one of them.

A central concept to CCL is the concept of *assembly*. An assembly in Cumbia is a group of concern specific model definitions, possibly conformant to different metamodels, and the information to relate those definitions. This information is enclosed in the CCL program for the assembly, which mainly consist on two kinds of instructions. On the one hand, there are instructions to establish relationships between the elements in model instances. On the other hand, there are the instructions to request the creation of instances of the models included in the assembly. At first, it may seem that the second kind of instructions are at odds with the stated goals for CCL, but these are necessary for two reasons. The first one is that it is a responsibility of the CCL program to describe how many instances of each model definition have to be created when the assembly is instantiated and the CCL program is run; in many cases, there is more than one instance per definition. The second reason is that not all the model instances have to be created when the assembly is instantiated. The number of instances may be even unknown at design type, and depend on the execution of the models. Therefore, the CCL program has to describe when to create instances of each model.

To understand what an assembly and a CCL program represent, consider the following example. *Learnflows* in a certain e-learning application involve two concerns: control and time. Thus, for each specific learnflow (e.g. online seminar on technology X) we have to create an assembly containing one control model, and several time models. The control model describes the learning activities that students have to perform. The time models describe time restrictions for the learning activities (e.g. deadlines). An assembly containing only model definitions is useless, because it does not contain any information about the relations between the model definitions. Therefore, a CCL program has to be included in each assembly, and this program has to contain the two kinds of instructions previously mentioned. On the one hand, it has to describe which model instances have to be created when the assembly is instantiated. For example, it may be the case that the same time restriction is used for two activities, and thus two instances of the same time model are required. On the other hand, it has to describe how to establish concrete links between elements in the model instances, using the interaction mechanisms of the open objects. This means that whoever wrote the CCL program had knowledge about the structure of the models, about the way in which they had to interact, and about the metamodels used to describe them.

Additionally, a parallel can be established between the previously mentioned elements of CCL, and elements of Conic, a coordination language where coordination is viewed as configuration [PA98, KMF90]. In Conic, there are typed*logical nodes* that have *exitports* that can generate messages, and *entryports* that can receive messages. Logical nodes are thus comparable to open objects that can generate and receive events. Besides offering the means to describe types of logical nodes (modules), Conic also provides instructions to describe system configurations. Just as in CCL, these instructions can be categorized as creation instructions, which describe the module instances to create, and linkage instructions, which describe the links to create between exitports and entryports in the module instances.

In the upcoming sections we present in detail the instructions available in CCL, and in section 5.4 we present a complete and detailed example. Additionally, section 5.3 explains how an assembly is packaged in a file with a special structure (a *cumbiar*) to be deployed in Cumbia.

5.2.1 Structure of a CCL program

Listing 5.1 shows the structure shared by every CCL program. The first part of it declares which model definitions are used in the assembly. This declaration specifies the names of the models to load, and their respective metamodels. To avoid ambiguities, models are fully identified with the name of the metamodel followed by the name of the model (e.g. `metamodel:model`). Moreover, to make CCL programs more readable, an alias is assigned to each model. These aliases are used in every point of the program where a model has to be referenced.

In the second part of the program, all the instructions to create model instances and to create links between their elements are found. These instructions are grouped in blocks, and each block is associated to an event that is relevant to CCL: when that particular event occurs, the block of instructions is executed with the adequate parameters. Currently, there are two types of events that can be used in CCL programs namely `Init` and `ModelInstanceCreation`.

The event `Init` signals the creation of the assembly. Therefore, the block of CCL instructions associated to this event is executed when the assembly is instantiated (see listing 5.2). The instructions found in this block usually serve to request the creation of model instances, but they can also create links between the elements in the newly created instances. Every CCL program must include

Listing 5.1: Basic elements of a CCL program

```
assembly {
 // Load and create aliases for the models used in the assembly
 load (MetamodelName1:ModelName1 ModelAlias1,
       MetamodelName2:ModelName2 ModelAlias2,
       ...
      );

 // References that can be used in any block
 global(ref1, ref2);

 // Block of instructions to perform when Event1 occurs
 on:Event1 ( parameters ) {
     ...
   }

 // Block of instructions to perform when an Event2 occurs
 on:Event2 ( parameters ) {
     ...
   }

   ...
}
```

a block of instructions associated to the event `Init`.

Listing 5.2: Block of instructions executed when the assembly is instantiated

```
 on:Init {
   ... // Instructions
 }
```

The other kind of event supported in CCL is `ModelInstanceCreation`. These events are generated when a new instance of one of the models in the assembly is created. In listing 5.3, there are two blocks of instructions associated to this event which differ on the model definition that is instantiated: the first block is executed when an instance of the model with alias `ModelAliasX` is created; the second block is executed when an instance of the model with alias `ModelAliasY` is created. In both cases, the new instance can be accessed using the name `localName` within the context of the block.

Listing 5.3: Blocks of instructions executed when model instances are created

```
on:ModelInstanceCreation (ModelAliasX instanceName) {
   ... // Instructions
}

on:ModelInstanceCreation (ModelAliasY instanceName) {
   ... // Instructions
}
```

5.2.2 Instructions to create model instances

The instruction to create an instance of a model in CCL can be used in any block of instructions. In listing 5.4 two model instances are created.

The first one (line 1) is an instance of the model that was declared with the alias `ModelAlias1`. In this code, a reference to the newly created model instance is stored in the temporary reference `localRef1` which can be used in the ensuing instructions. If the variable is declared to be global, then it can be used in other blocks.

The second part of the listing (lines 2 to 5) show an alternative way of creating a model instance that stores some information in the memory of the instance. By offering this, it is possible to parameterize the instances created, and make distinctions between them. In order for this information to be useful for the model execution, the names of the parameters must match the names of the data that the model expects to find in its memory.

Listing 5.4: CCL Example: Model instantiation

```
1 localRef1 = new ModelAlias1;
2 localRef2 = new ModelAlias2 {
3    param = value;
4    ...
5 };
```

5.2.3 Instructions to create links

The last aspect of CCL are the actual operations to create relationships between elements located in different model instances. Since the elements involved are all open objects, these relationships take advantage of the interaction mechanisms of the open objects. These mechanisms were described in section 4.2.2.

The first kind of relationships that can be created with CCL is based on the introduction of actions. Listing 5.5 shows how to create one of those.

Besides declaring the type of link to create (`createActionLink`), the first line also specifies the name of the link (“link_name”). The Cumbia Weaver stores the information about the links created, and this name can later be used to navigate or modify the links.

The following lines in the listing define and configure actions to install in specific transitions. In line 2, the action installed is of a predetermined type which can be configured to call specific methods in other open objects. An instance of this action is installed in the transition called *activation* of the element described with the expression `fcpInst["#root"]`. In this expression, `fcpInst` points to a model instance, while `["#root"]` is a navigation expression that points to elements in that instance. The new action is configured to call the method `notifyStart( )` in an element of the model instance identified with `tr`.

Lines 3 and 4 of listing 5.5 present an alternative way to specify the action to install in the transition. In this case, the action is not the predefined one but a specific one provided with the assembly (e.g. `"cumbia.ccl.test.ResetAction"`). The location where this action is added is described as in the first case, but its specific behavior depends on its implementation. The action can also be configured with a parameter. In this case it receives as parameter a reference to the root element in the `tr` model instance.

Listing 5.5: CCL Example: Action Link

```
1 createActionLink ("link_name") {
2   fcpInst["#root"]|activation -> tr["#root"]::notifyStart();
3   fcpInst["#root"]|deactivation ->
4       createAction("cumbia.ccl.test.ResetAction", tr["#root"]);
5 } ;
```

The other kind of relationships that can be established with CCL is based on events rather than actions. In this case, instead of adding new actions, the idea is to associate an existing transition of an element to an event generated by another element. Nevertheless, in Cumbia transitions can only be triggered by *one* event. Thus, the specified transition is duplicated and the new one is associated to the event.

Listing 5.6 shows how a relationship based on events is created in CCL. As in the previous case the first line names the link. Line 2 first specifies the event and then it specifies the transition to be triggered by that event. This transition has to be an existing transition which is duplicated when `createEventLink` is executed. The only differences between the original transition and the new one are the trigger event (which in the second case is the event specified), and the name of the transition (which in the second case has a consecutive number appended).

Listing 5.6: CCL Example: Event based Link

```
1 createEventLink ("link_name") {
2   fcpInst["#root"]:starting -> tr["#root"]|beginning ;
3 };
```

The two types of relations discussed form the core of the composition in Cumbia. Since they are independent of the metamodels they can be easily applied to any concern specific model. Furthermore, they do not have a destructive impact on the elements of the models: these relationships can be easily removed to restore the elements to their original state.

5.3 An architecture to execute concern specific models

In section 4.5 we presented the base architecture of workflow engines based on Cumbia. However, in that section we only considered the case of engines that use a single modeling language. Therefore, these engines depend on only one metamodel and it is not necessary to do any kind of model composition.

This section complements what was introduced in 4.5 by presenting an architectural view of Cumbia based workflow engines that support multiple concern specific workflow languages. This section is divided in two parts. In the first one, we present the general architecture and relate it to the workflow reference model of the Workflow Management Coalition. The second part focuses on the Cumbia Weaver which is a central component in the whole architecture as it relates the concern specific engines.

5.3.1 General architecture

The architecture of a Cumbia based workflow solution which supports multiple concern specific workflow languages is shown in figure 5.2. This architecture includes the following components and interfaces.

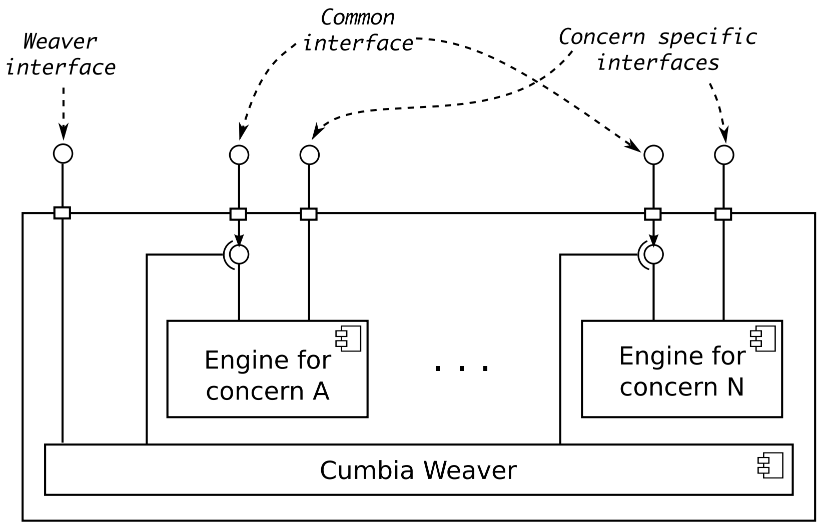


Figure 5.2 Architecture of a workflow solution based on Cumbia

- **Concern specific engines**

  In order to run models conformant to different metamodels it is necessary to have one engine for each metamodel. The reason for this is that each engine uses one instance of the Cumbia Kernel, and that each instance of this kernel can be configured with one metamodel at a time. Each engine holds the same responsibilities it would if there were not other engines.

  An important characteristic of the architecture is that each of these engines do not need to know each other in order for their models to interact. This is because interaction is achieved by the relationships created by the Cumbia Weaver. This characteristic facilitates the reuse of these engines, and makes it more easy to replace one engine or to add a new one depending on the concerns involved in each context.

- **Common interfaces**

  The various engines depicted in the figure usually offer two separate interfaces. The first one is an interface shared by all the engines, which corresponds to the interface offered by the Cumbia Kernel. This interface can be used by external applications to interact with each engine using the same protocols. Furthermore, this interface is used by the Cumbia Weaver to interact and require services from the engines. For instance, the Cumbia Weaver uses this interface to load models into each engine, request the creation of model instances, and access the elements inside those instances.

- **Concern specific interfaces**

  In addition to the common interface, each engine can also offer an ad-hoc interface where services specific to that engine are exposed.

- **Cumbia Weaver**

  The Cumbia Weaver has two main responsibilities in this architecture. The first responsibility is that of interpreting CCL programs to establish relationships between model instances. This is possible because the Cumbia Weaver has also the responsibility of locating and maintaining references to the engines involved in the assemblies. Therefore, the Cumbia Weaver has to be used to locate or access each engine.

- **Weaver interface**

  The other important interface in this architecture is the interface of the Cumbia Weaver. Through this interface it is possible to load new assemblies or to request their instantiation. Furthermore, this interface also serves to query the instances of the assemblies, and to query and analyze the relations established between elements in the model instances involved.

The Workflow Management Coalition proposed in the Workflow Reference Model (WRM) a series of five interfaces that, according to them, have to be offered by every workflow engine [Hol95, Hol04]. We now analyze how the 5 interfaces appear in the Cumbia architecture that was just presented. Figure 2.1 shows the 5 interfaces.

Interface 1: Process definition tools

In the WRM, this interface is used by process definition tools (editors) to deploy workflows into engines. In the case of Cumbia, two things have to be considered. In the first place, it is possible to build editors capable of deploying workflows into the Cumbia engine through the usage of the Cumbia Weaver interface. We have not built such tools yet, but there are no restrictions that would prevent it.

The second thing to consider is that the Cumbia Weaver is also capable of loading a kind of package that we have called *cumbiar*. A *cumbiar* file is similar to a *jar* file [Ora10a] because it is compressed, has a meaningful internal structure, and has a predefined descriptor. In the case of a *cumbiar*, the internal structure depends on the metamodels and models included in it, and the descriptor is a CCL program. The relevance of this format is that process definition tools can use it to export data in a way that can be directly used by the Cumbia Weaver.

Interface 2: Workflow Client Applications

The second interface in the WRM is used by applications that mediate in the interaction between users and running workflows. For instance, one such client application can request information from a user and then give that information back to an activity in order to process it and continue with the workflow execution.

Client applications can interact with Cumbia based workflows by means of the two kinds of interfaces previously described: the common interface, and the concern or engine specific interfaces. In the first case, the main advantage is that applications are more easily reused. In the second case, the main advantage is that the interaction can be more powerful since the interfaces are, in theory, more expressive.

Interface 3: Invoked Applications

This interface in the WRM is used to allow back and forth interaction with external applications that participate in a workflow. For instance, in the case of a BPEL engine this is the interface that allows the invocation of web-services via SOAP, and the reception of responses to asynchronous messages.

In the case of Cumbia, the elements in each metamodel can communicate with external applications through any means they want. There is not a single outgoing communication channel to be shared by all engines. With respect to incoming messages, each engine can setup ad hoc reception channels as part of engine specific interfaces. If necessary, external applications can locate these engines by using the services provided through the weaver interface.

Interface 4: Other Workflow Enactment Services

In the WRM, interface 4 is described as the interface to allow the interaction between several instances of a workflow engine, or between several different workflow engines. In Cumbia, we have not attempted to provide specific mechanisms to support this.

Interface 5: Administration and Monitoring Tools

The fifth and final interface described in the WRM allows the interaction of administration and monitoring tools with the engine. These tools have two main objectives. The first one is to control and manage the behavior of the engines and the execution of the workflows. The second one is to gather and display information about the running workflows.

In the case of Cumbia, this kind of tools can use the mechanisms previously discussed and, in particular, those described for interface 2 of the WRM.

5.3.2 The Cumbia Weaver

With respect to model composition and coordination, the Cumbia Weaver is the central element of the architecture. As it has been briefly mentioned in previous sections, the Cumbia Weaver has several responsibilities from the loading of an assembly (for instance, as a *cumbiar*), to the removal of finished instances. In this section we analyze all these responsibilities.

Assembly load

The first responsibility of the Cumbia Weaver is to load assemblies so they can be later instantiated. This requires the following intermediate steps:

- *Handle model resources.* The majority of the resources required to execute Cumbia models should be part of the engines. For instance, the implementation of the entities of the used open objects is normally part of the resources that are used to configure the Cumbia Kernel in an engine. Nevertheless, since each model can extend its metamodel by using the extension mechanisms presented in chapter 4, each model can require additional resources (configuration files, images, keys, etc.). The Cumbia Weaver has the responsibility of loading those additional resources and pass them to the specific engines that have to use them, before instances of those models are created.

- *Handle model definitions.* The Cumbia Weaver has to load the definitions of the models used in an assembly, and load them in the adequate engines.

- *Load CCL resources.* The CCL code that describes the relationships in an assembly may require additional resources. In particular, each CCL program can use ad hoc actions that have to be supplied with the assembly. The responsibility of the Cumbia Weaver is to load these resources and make them available when the CCL code is executed.

- *Load the CCL program.* The Cumbia Weaver has to load the CCL program and have it ready to be executed when instances of the assembly are created.

After these steps are completed it becomes possible to create instances of the assemblies. This whole process does not necessarily have to be done step by step: from the perspective of a user, it can be done in a single step where a *cumbiar* file is loaded. Section 5.4.5 will present the structure of a *cumbiar* with a concrete example.

Instance creation

The creation of an instance of an assembly has two relevant steps. In the first one, a space to store the state of the assembly is created inside the Cumbia Weaver and it receives an identifier. In this way it is possible to locate and query the existing instances, and there is a place to group all the information about the run time state of the assembly. Figure 5.3 presents this graphically.

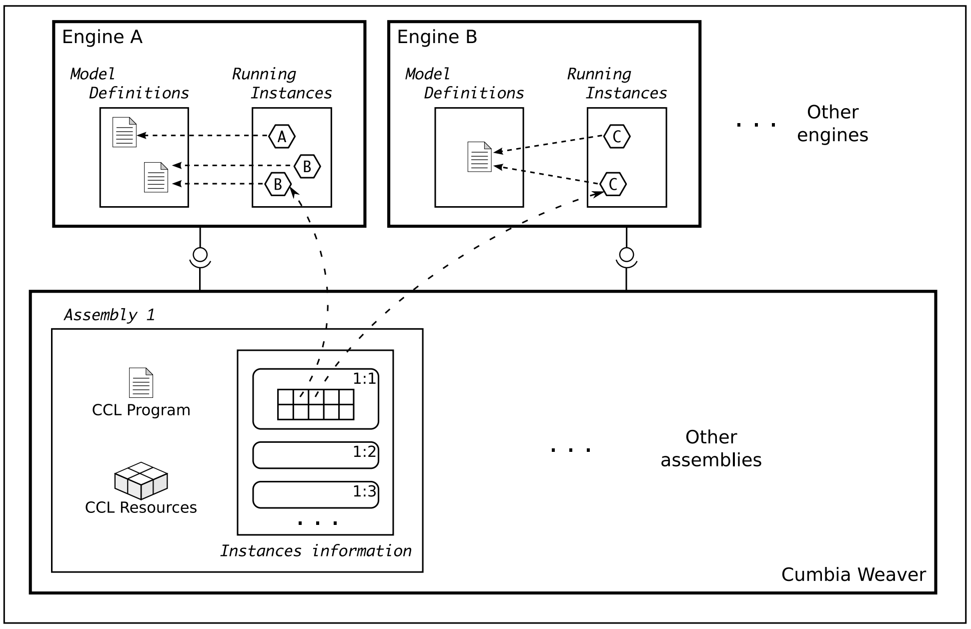


Figure 5.3 Representation of the run time state of the Cumbia Weaver and of two engines

The lower part of the figure offers a run time representation of the Cumbia Weaver. For each assembly, there is an area where the CCL program and the CCL resources are stored. This space is also used to maintain run time information about instances of the assembly, using a unique identifier for each one (e.g. 1.1, 1.2, etc.). The state of each instance includes the state of the variables, and references to model instances. In the upper part of the figure engines and their relevant elements are depicted. These include the definition of the models (which are referenced in the CCL program), and the instances of the models which are created as required in the CCL program.

The second step in the creation of an assembly is the interpretation of the CCL program, which can also be subdivided in several steps.

1. The first step is to verify the existence of the necessary model definitions as declared in the `load` part of the CCL program. Furthermore, the information about the aliases is stored in the assembly space to be used in the rest of the CCL program execution.

2. The second step is to start executing the block of CCL instructions associated to the `Init` event.

3. Finally, the third step is to execute the other blocks of CCL instructions whenever the corresponding events occur.

The CCL instructions available can be divided in two main groups. On the one hand, there are the instructions to create instances of the models, while on the other hand there are the operations to establish relationships between elements in the instances. In the former group, an important detail is that the Cumbia Weaver does more than just requesting the corresponding engine to create the instance. It also installs the necessary listeners to receive other events generated by the engines and by the Runtime Controllers.

The rest of the CCL instructions are related to linking the elements in the model instances, and are achieved by employing the operations offered in the interface of the Cumbia Kernel. Furthermore, these operations also update a registry of the relations created as part of each assembly instance, and of the elements involved in each relation.

All the tasks performed by the Cumbia Weaver that were just described depend on the fact that it needs access to the engines involved. In particular, it needs to be able to interact with them to request the loading and instantiation of models.

5.4 A sample workflow

This section presents a complete workflow that involves two concerns with mutual dependencies at run time. The goal of this section is to show how an assembly is defined and described, how it is instantiated, and finally how its models are composed and their executions are coordinated.

This section starts with a presentation of the concerns and the metamodels involved (`MiniBPMN` and `XTM`). Then, the models are presented: first the control model, and then the models that impose time restrictions on the control. Next, we present the CCL program that describes the coordination between these models. Finally, we describe the weaving process and the execution of these models.

5.4.1 The metamodels

For the sample workflow that we are discussing in this section we are going to use two metamodels that describe two co-dependent concerns. The first concern is the control concern, and we are going to use `MiniBPMN` to build control models.

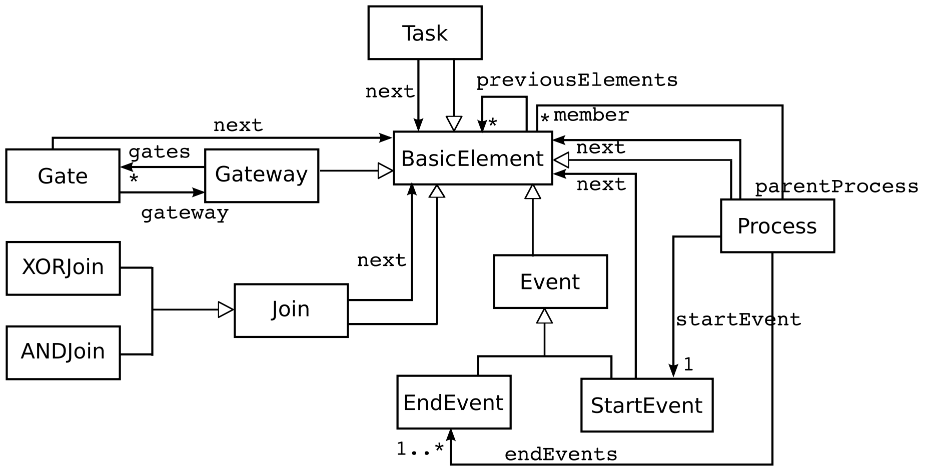


Figure 5.4 `MiniBPMN` metamodel

The elements of this metamodel are depicted in figure 5.4. This metamodel was fully presented in chapter 4 and we retake it here in an unaltered form.

The second metamodel is called `XTM`, and it is used to describe models representing time restrictions that workflows *should* adhere to. The acronym `XTM` stands for Extensible Time Metamodel, and its main elements are depicted in figure 5.5. The central element of this metamodel is called `TimeRestriction`, and it encapsulates the logic to decide whether a time restriction has been fulfilled or not. `XTM` supports many types of time restrictions, and this can be seen in the hierarchy of extensions to `TimeRestriction`. In the figure we have only included the three kinds of restrictions that are relevant to the sample workflow (`DurationLowerLimit`, `DeadlineLowerLimit`, and `PeriodicTaskStopEvent`), but the complete metamodel includes fourteen of them.

In order to keep track of time elapsed or to check whether a deadline has passed, `XTM` includes different kinds of timers that generate event notifications when certain conditions are valid (a deadline has passed or a time lapse has passed, for example). These notifications are needed by the `TimeRestriction` to update its state.

On the other hand, the time restrictions that can be modeled with `XTM` are not *hard constraints* on the execution of other models. `XTM` cannot enforce the conformance to the time restrictions. Instead, it can provide sets of activities to execute depending on whether each restriction is respected or not. This means that it is possible to define some actions to execute if a restriction is fulfilled, and some other actions to execute if the restriction is not fulfilled. These actions are grouped in `TaskSets`, and the execution of each task set is triggered by a different event notification emitted by the `TimeRestriction`.

The last element in this metamodel is called `EntryPoint`. An `EntryPoint` works as an adapter to elements in other models which are subject to time restrictions. `XTM` models are usually composed by an `EntryPoint`, a `TimeRestricition`, and then the `Timers` and `TaskSets` that the restriction requires. The run time state of the `TimeRestriction` depends on the events received from the timers and the `EntryPoint`. In turn, the events generated by the `EntryPoint` depend on event notifications or methods invoked by elements in other models. Therefore, when `MiniBPMN` and `XTM` are used together, the `EntryPoints` are normally

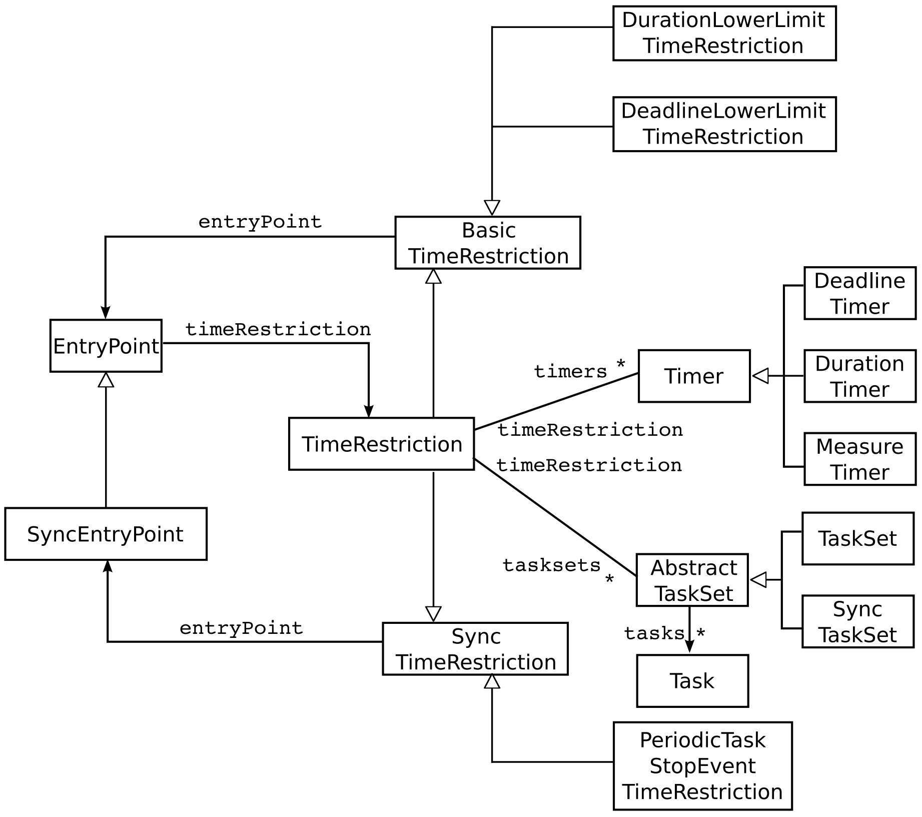


Figure 5.5 Fragment of the `XTM` metamodel

the only elements of `XTM` woven to elements in other models.

In `XTM`, the role of the `EntryPoint` can be assimilated to the role of a *façade*, because it is the sole point of communication between the control model and a time restriction. The `EntryPoint` also works as an *adapter*, because it converts events and calls made by external elements into events that are expected by the `TimeRestriction`. However, this pattern does not need to be followed in every metamodel. Depending on each particular situation it can also be acceptable to have many points of contact between models.

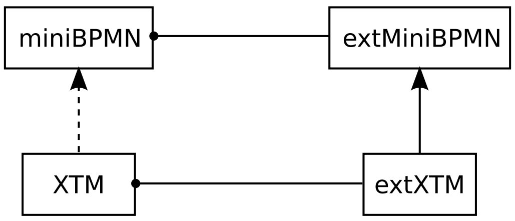


Figure 5.6 Relations between the metamodels in the sample workflow

Figure 5.6 shows the relations between the metamodels involved in the sample workflow. The left part of the figure shows the relationship between `MiniBPMN` and `XTM`: models built with `MiniBPMN` are oblivious of `XTM` models, while `XTM` models observe, react, and can perform actions on `MiniBPMN` models. The line between `XTM` and `MiniBPMN` is not continuous because `XTM` models can be composed to models built with many other control metamodels (e.g. BPMN, BPEL, or YAWL). More concretely, this means that the implementation of `MiniBPMN` is not aware of the presence of any other metamodel: `MiniBPMN` models can thus be used independently of other models. Conversely, `XTM` models always require the presence of at least another model, regardless of its metamodel.

The right part of the figure shows that extensions to some metamodels do not have to be related in the same way as the base metamodels. For example, the relationship between the extensions in this case is continuous because specific tasks included in the `XTM` extension (`extXTM`) depend on specific elements of `extMiniBPMN`.

In the next sections we will present the control and the time restriction models for the sample workflow. Furthermore, we will provide a few more details about `XTM`'s elements, including presenting the state machines of some of its elements.

5.4.2 The control model

The control model for the sample workflow is the same that was presented in section 4.7. This model, which is shown again in figure 5.7, represents the control flow of a process in the context of financial services. This process was modeled with `MiniBPMN` but, unlike the previous chapter, in this one the model is not going to be executed by itself. Instead, it is going to be complemented with time restrictions and both the control model and the time models will be executed in a coordinated fashion.

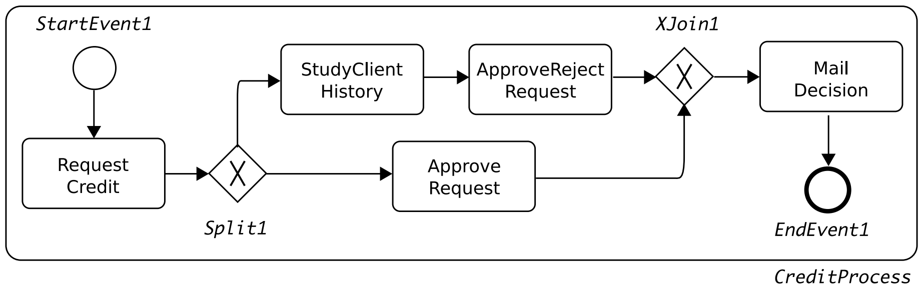


Figure 5.7 The control concern in the sample workflow

5.4.3 The time models

In this section we will describe the concern of the sample workflow that establishes time restrictions and some actions to execute which depend on those restrictions. These restrictions are defined in three models that are conformant to the `XTM` metamodel that was previously introduced. Figure 5.8 illustrates the idea of how the three time restrictions (TR1, TR2, and TR3) are going to be connected to different elements of the `MiniBPMN` model.

We now describe these three models and in the following sections we will show in detail how these models are composed to the `MiniBPMN` control model. In those sections we will also present in more detail the `XTM` elements involved, including the state machines most relevant to the composition.

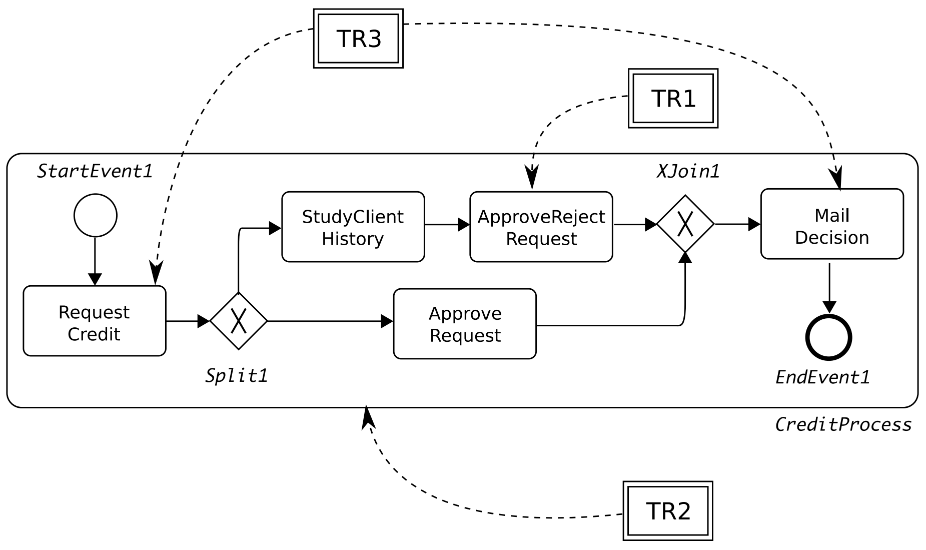


Figure 5.8 Integration between control and time in the sample workflow

Time Restriction 1: maximum time for approval or rejection

The first restriction states that the task `Approve/Reject Request` should not last longer than one day. Furthermore, this restriction specifies what to do when that restriction is not observed. In such cases, an email must be automatically sent to a supervisor of the employee that took too long to answer.

Figure 5.9 shows an object diagram of the model for this restriction. The kind of time restriction employed (`Duration Lower Limit Time Restriction`) requires the usage of a `DurationTimer`, which emits a signal when a programmed time lapse has elapsed. This time restriction coincides with a time restriction pattern where the interval between two events in the workflow is not expected to be superior to a certain amount of time.

This model only includes one `TaskSet` because the time restriction only specifies what to do when the duration is exceeded. Nevertheless, the characteristics of `XTM` and, especially, the structure of the state machine of the `TimeRestriction` makes it also possible to specify something to do when the duration is *not* exceeded (see figure 5.12).

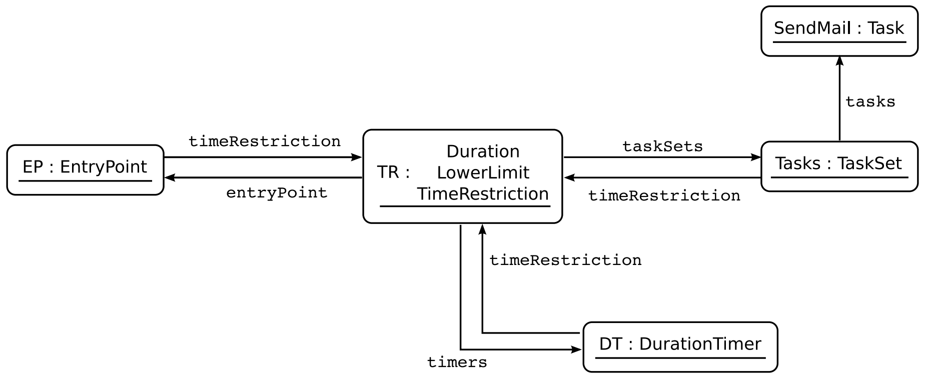


Figure 5.9 Objects diagram of the first time restriction

Time Restriction 2: unfinished processes at the end of the month

The second XTM model is used to gather a list of the processes that are still unfinished at the end of this month. In order to do so, the model shown in figure 5.10 uses a TimeRestriction of the type Deadline Lower Limit Time Restriction. This kind of restriction corresponds with a time pattern where an event in the workflow must occur before a given deadline. In this case the DeadlineTimer is used to generate an event when the end of the month is reached.

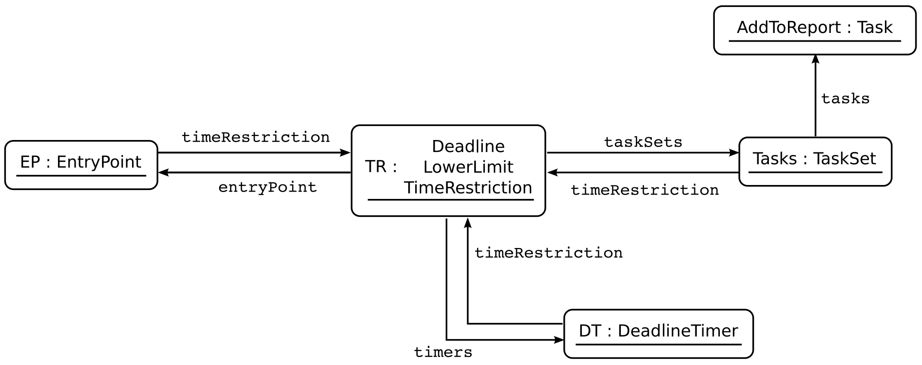


Figure 5.10 Objects diagram of the second time restriction

In this case, there is also only one TaskSet and the only action in it is used to add the identifier of the process to the report of unfinished processes.

Time Restriction 3: daily status report

Finally, the third restriction specifies that a daily report with the status of the whole process has to be sent to each client while the process is still unfinished and he has not received a notification. In this case, the TimeRestriction is of a type that is slightly different to the previous ones because it can execute each TaskSet different times, once for each time the MeasureTimer emits a signal (see figure 5.11). In this case, the only Task in the model is used to send the report to the client.

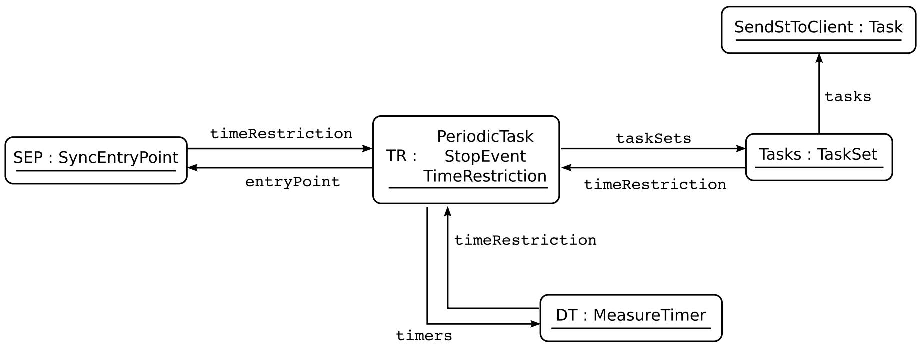


Figure 5.11 Objects diagram of the third time restriction

5.4.4 The composition code

In the previous two sections we presented the MiniBPMN and the XTM models of the sample workflow. However, up to this point there are no real relations between those models. At most, there are only intentions in the descriptions of the `XTM` models which refer to specific activities in the control model.

In this section we materialize these relations using CCL. To facilitate the description of the CCL code, it will be divided and presented in four parts.

Assembly declaration and initialization

Listing 5.7 shows the first part of the CCL code for the sample workflow. This code describes two main aspects of an assembly.

In the first place, it declares the models that are going to be part of the assembly, and assigns a model alias to each one. The listing shows how this particular sample assembly is going to include one `MiniBPMN` model named ‘fastCreditProcess´, and three `XTM` models. The code also declares `fcpInst` as a global variable and thus it becomes usable in every part of the code.

Listing 5.7: Assembly declaration and initialization of the assembly instance

```
1 assembly {
2  load (miniBPMN:fastCreditProcess FCP,
3         XTM:restriction1 Res1,
4         XTM:restriction2 Res2,
5         XTM:restriction3 Res3 );
6
7  global (fcpInst);
8
9  // Initialization of an assembly's instance
10 on:Init {
11   fcpInst = new FCP;
12   new Res1;
13   new Res2;
14   new Res3;
15   }
16   ...
17 }
```

The second part of the code (lines 9 to 15) specifies what has to be done when the assembly is instantiated. First of all, line 11 specifies that a new instance of the `MiniBPMN` model has to be created. A reference to the newly created model instance is stored in the global variable `fcpInst`. The other lines in this block create new instances of the `XTM` models. However, because of the event based organization of the CCL blocks, it is not necessary to store references to these instances.

Composition of the first time model

The composition of the `MiniBPMN` model with the `XTM` model corresponding to the first time restriction is determined by the CCL code shown in listing 5.8. This block of code is associated to the event of the creation of an instance of the `XTM` model.

One thing to note about the code in listing 5.8 is that in this case the elements in the `MiniBPMN` model are not selected with an expression of the navigation language. Instead, the function `findByName` is used to select the elements using their name, which is unique within each model instance.

Listing 5.8: Composition of the first time model

```
on:ModelInstanceCreation (Res1 newTR) {
  createActionLink ("Approval/Rejection Duration") {
    findByName(fcpInst, "ApproveRejectRequest")|activate ->
                 newTR["#root"]::notifyIntermediate();
    findByName(fcpInst, "ApproveRejectRequest")|deactivate ->
                 newTR["#root"]::notifyEnd();
  };
}
```

The result of executing this CCL code is represented in figure 5.12[2]. Two additional actions are installed into transitions of the state machine of the task `ApproveRejectRequest`. Furthermore, these actions are configured to invoke the methods `notifyIntermediate( )` and `notifyEnd( )` of the `EntryPoint` of the `XTM` model.

Composition of the second time model

The second `XTM` model is composed to the `MiniBPMN` model by using a different kind of composition mechanism. While the previous example used additional actions, this one works by subscribing one element (the entry point in the `XTM` model) to an event generated by one element in the other model (the process in the `MiniBPMN` model). Listing 5.9 shows how this is achieved.

Listing 5.9: Composition of the second time model

```
on:ModelInstanceCreation (Res2 newTR) {
  createEventLink ("End of Month Report") {
    fcpInst["#root"]:deactivation -> tr["#root"]|endTransition;
  };
}
```

Figure 5.13 shows the end result of this composition: in this case, a new transition triggered by the event `deactivation` is created.

Composition of the third time model

The final part of the CCL program (listing 5.10) is used to compose the third `XTM` model to the `MiniBPMN` model. The main difference between these cases and the previous ones is that two different elements of the `MiniBPMN` model are woven to a single element of the `XTM` model. In the listing shown below, an action is first installed in a transition of the `Task` 'RequestCredit', and then a second one is installed in a transition of the `Task` 'MailDecision'. Figure 5.14 shows the result of this composition.

[2]In figures 5.12, 5.13, and 5.14 we have left out various details in order to present in the figures only the elements most relevant to the composition.

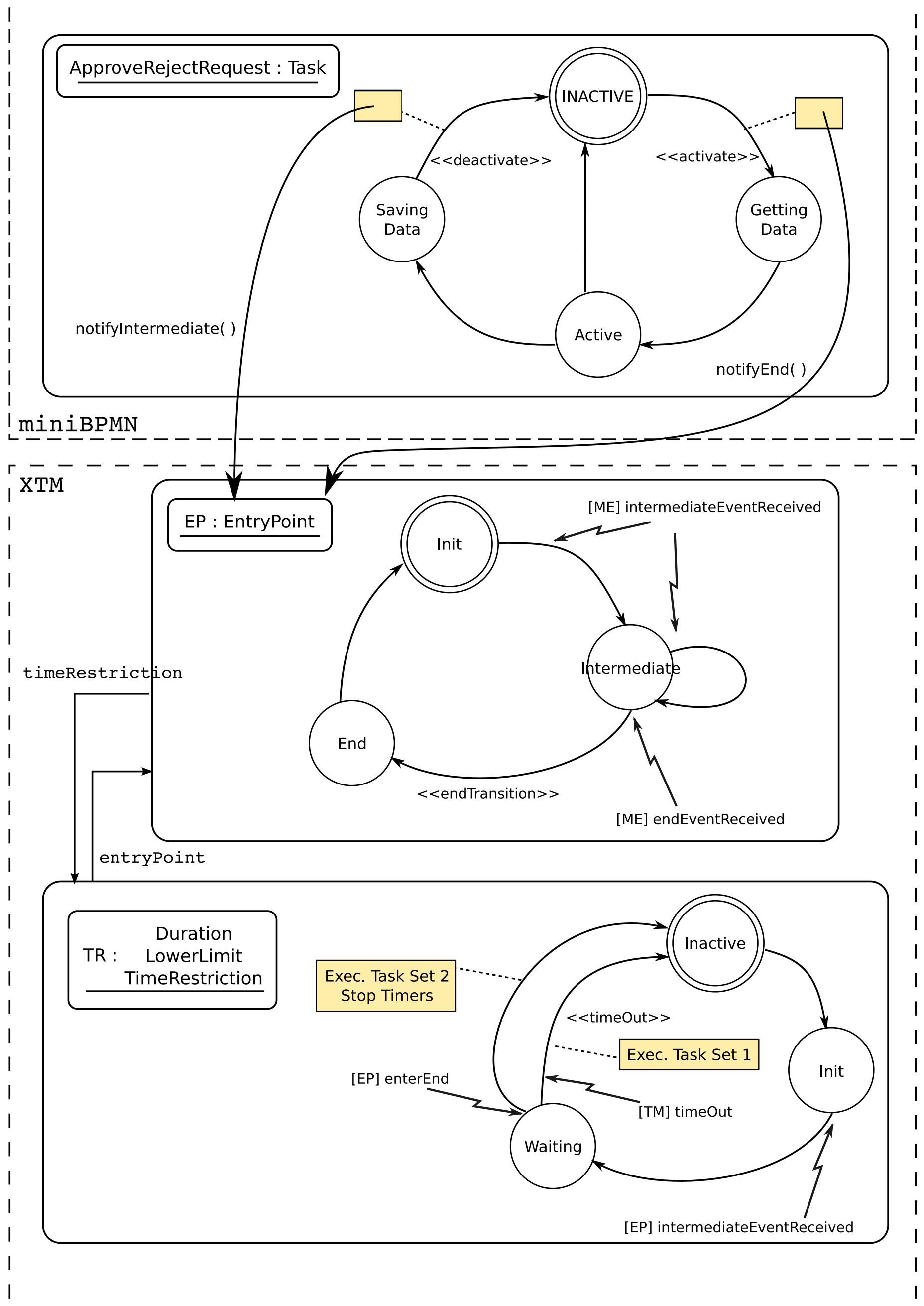


Figure 5.12 Partial view of the `MiniBPMN` model composed with the first `XTM` model

Listing 5.10: Composition of the third time model

```
on:ModelInstanceCreation (Res3 newTR) {
   createActionLink ("Daily Report Links") {
     findByName(fcpInst, "RequestCredit")|activate ->
                  newTR["#root"]::notifyIntermediate();
     findByName(fcpInst, "MailDecision")|deactivate ->
                  newTR["#root"]::notifyEnd();
   };
}
```

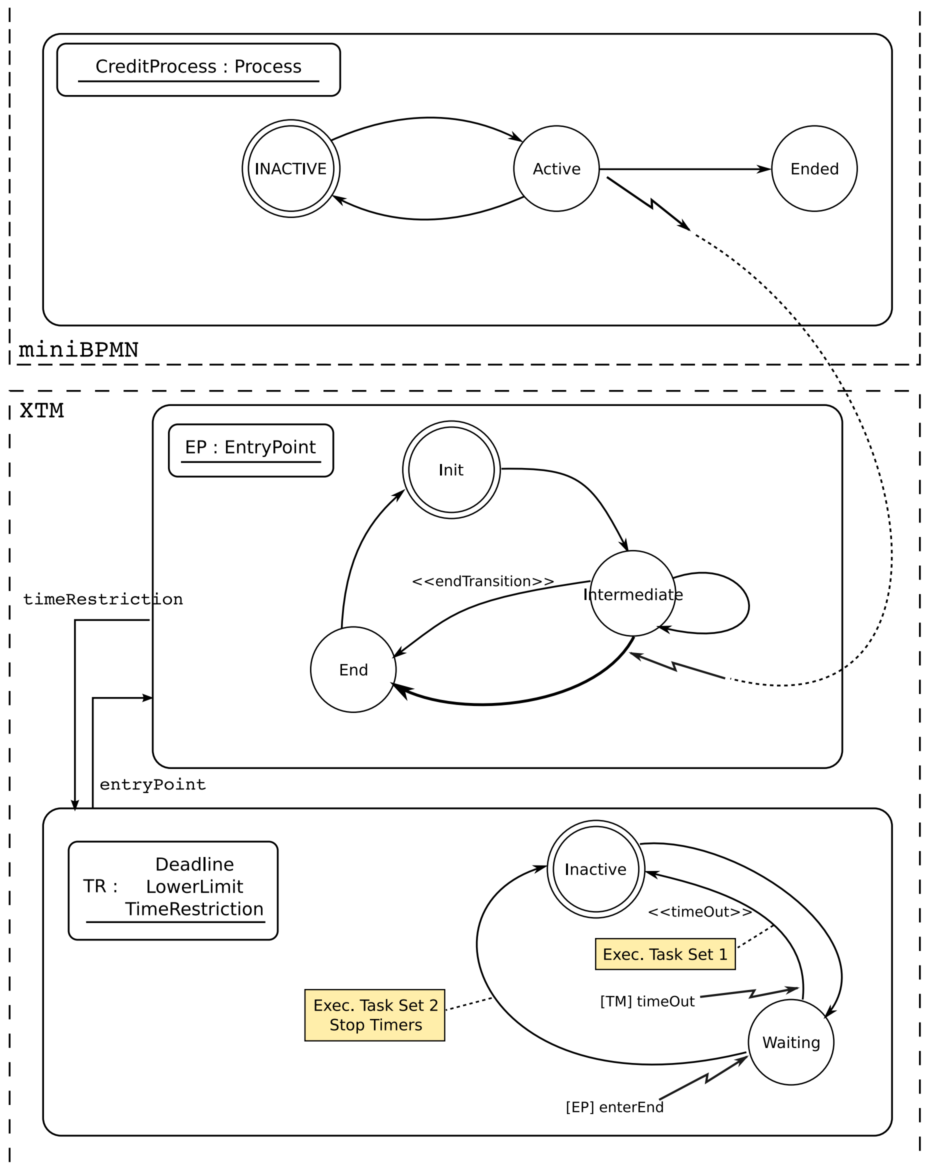


Figure 5.13 Partial view of the `MiniBPMN` model composed with the second `XTM` model

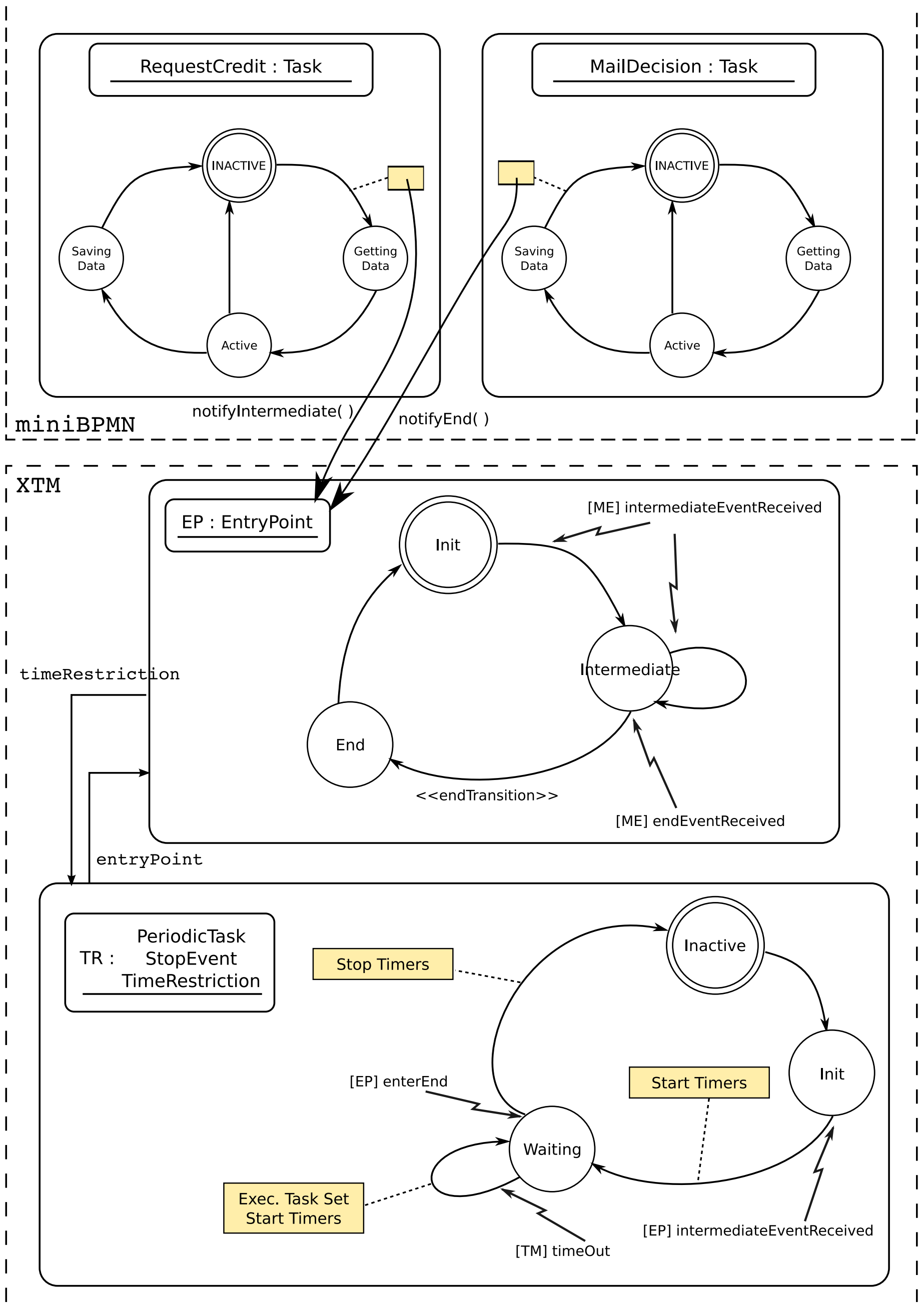
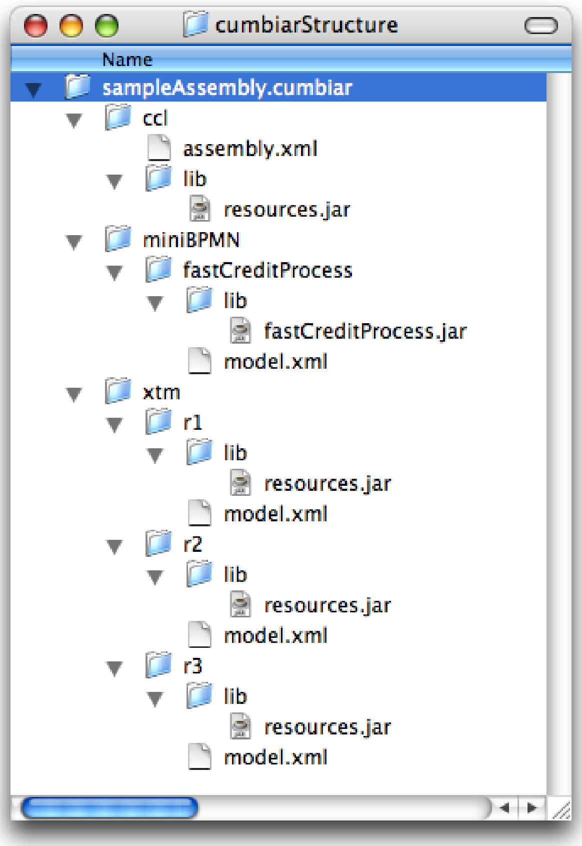


Figure 5.14 Partial view of the `MiniBPMN` model composed with the third `XTM` model

5.4.5 Packaging and using the assembly

The step to perform after building the models and writing the CCL program to compose them, is to package the assembly in a *cumbiar* file. For the sample workflow, the *cumbiar* file has the structure shown in figure 5.15. For each model in the assembly, this file includes the description of the models, and the additional resources that are required in each one. Similarly, this file also includes the CCL program (`assembly.xml`) and its additional resources (in this example this file is empty as there are no additional CCL resources required).

Cumbiar files are loaded in the Cumbia Weaver, which analyses their structure and the CCL program, and loads the resources and models in the adequate



```
▼ sampleAssembly.cumbiar
    ▼ ccl
        assembly.xml
        ▼ lib
            resources.jar
    ▼ miniBPMN
        ▼ fastCreditProcess
            ▼ lib
                fastCreditProcess.jar
            model.xml
    ▼ xtm
        ▼ r1
            ▼ lib
                resources.jar
            model.xml
        ▼ r2
            ▼ lib
                resources.jar
            model.xml
        ▼ r3
            ▼ lib
                resources.jar
            model.xml
```

Figure 5.15 Structure of the *cumbiar* file with the sample assembly.

engines. In the example considered, this means loading the fastCreditProcess in the `MiniBPMN` engine, and loading the three time restriction models in the `XTM` engine, each with their respective resources.

After performing this step, it becomes possible to request the creation of instances of the assembly. When such a request is issued, through the interface exposed by the Cumbia Weaver, the block of instructions associated to the `Init` event is executed. In this case, the instructions require the creation of instances of the models inside each engine. Since every engine generates some events informing about the creation of those instances, the blocks of instructions associated to the events `ModelInstanceCreation` are also executed, one by one.

After these steps are completed, the end result is an instance of the assembly composed by a collection of instances of the models, whose elements are related as the CCL program specifies.

5.5 Summary

To conclude this chapter we present an evaluation of the composition strategy of Cumbia, based on the reference process framework discussed in the previous section. The following is an overview of the composition strategy of Cumbia from each point of view.

- **Ordering**

The order to perform the composition depends on the order of CCL instructions: the Cumbia Weaver executes blocks of instructions associated to an event. Within each block, instructions are executed one by one, in the order specified in the file.

- **Ordering Flexibility**

  There is flexibility in the order of the composition because the blocks of instructions are associated to high level events produced by the workflow engines. In general, a fixed order cannot be predicted, but there is no concurrency inside the Cumbia Weaver.

- **Stopping Criteria**

  For each event, there is a number of instructions and the composition ends when those instructions are all executed. The whole composition process finishes when no more events occur.

- **Stopping Criteria Flexibility**

  The flexibility is given by the usage of events to group and organize the execution of the CCL composition instructions.

- **Contribution**

  In CCL the contributions are concern specific models.

- **Contribution Granularity**

  The granularity of the contributions in CCL is at the level of elements in a specific model instance.

- **Contribution Selectivity**

  The selection in CCL is arbitrary and any element in a model definition can be selected to be composed.

- **Location Description**

  In CCL the locations are described using expressions based on the navigation language presented in section 4.2.3. Using the information present in model definitions, the locations are described with respect to specific model instances.

- **Location Freedom**

  In CCL, the valid locations are all the elements in a model instance.

- **Combination Description**

  In our strategy, the combination is described by writing CCL programs.

- **Combination Expressiveness**

  The expressiveness of the combination is limited by the expressiveness of the available CCL operations.

- **Translation**

  We do not use a translation step in our composition strategy because it does not need models to be all conformant to the same metamodel.

- **Fastening**

  The fastening is done by the Cumbia Weaver when it executes the actual CCL operations. This fastening is done between elements of model instances.

- **Symmetry**

  Our composition strategy is symmetric: every model is at the same level. Furthermore, since all the original models are kept and are distinguishable after the composition, it is not possible to speak about a base model that serves as the base for the final composed model.

Chapters 4 and 5 have presented the core details about the Cumbia platform, and have illustrated its usage with a simple application. Nevertheless, these chapters have not studied practical considerations that are relevant for the development of such an application. For example, not a lot has been said about the process to design or select concern specific languages, about the roles played by the people that participate in the development, or about the complementary tools that they can use to facilitate the development. Chapter 6 focuses on providing all this practical information.

6 Towards a development process for workflow engines based on Cumbia

In previous chapters we have presented the Cumbia platform and its base elements (the open objects). In those chapters, we have only discussed the tools that are central to the platform: the Cumbia Kernel and the Cumbia Weaver. Similarly, we have focused on the central tasks of designing and implementing metamodels and models. However, we have not shown in detail the relationships between these tasks, and we have not discussed any of the complementing tasks.

This chapter addresses these topics. It describes all the stakeholders involved in the process of designing, implementing, and using concern specific workflow languages with Cumbia. Then, this chapter analyses the tasks that each stakeholder performs. To give these tasks a context and logical relations, we have organized them in a software development process to produce workflow engines. Finally, this chapter also presents the supplementary tools that we have already developed to support some of the aforementioned tasks.

The chapter begins with a rapid presentation of the whole development process and of the relevant stakeholders, and then it describes each task, artifact, and tool in more detail.

6.1 Process overview

Developing, maintaining, and evolving workflow languages and engines requires a complex process involving several small activities of different natures. Some of these activities have a strong technical component, and thus they have to be performed by people with technical skills. On the other hand, other activities are more suitable for people with a deep knowledge of the particular domains. In this chapter, we present a rudimentary development process that goes from the design of the concern specific workflow languages, to the execution and monitoring of workflow assemblies. Besides presenting the specific activities to be performed, we also discuss the artifacts produced and consumed by each activity, the spectrum of participants involved, and the tools that these participants require in each step. Figure 6.1 uses BPMN's notation to graphically represents the five main groups of activities (phases) in the process.

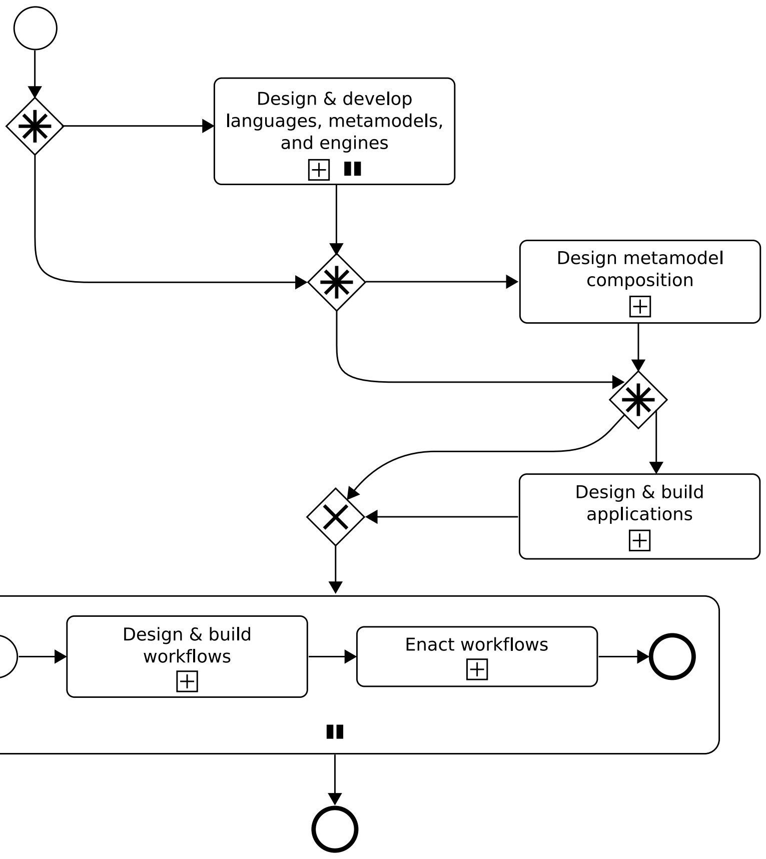

Design & develop languages, metamodels, and engines
Design metamodel composition
Design & build applications
Design & build workflows
Enact workflows

Figure 6.1 The five main phases in the process for developing workflow languages and engines based on Cumbia

The five phases identified can be summarized as follows:

1. **Design & develop languages, metamodels and engines.** Based on requirements from specific domains, workflow languages are designed. Since the languages can be concern specific (CSWfLs), this can be done several times, in parallel. The metamodels and engines corresponding to each language are designed, built, and tested in this step. In figure 6.1 this step is shown as optional because it is possible to reuse, in new applications, languages developed in previous iterations. In such cases the goal of the process becomes integrating those languages under a new workflow application.

2. **Design metamodel composition.** The possible interactions between specific groups of concern specific languages are analyzed. These interactions are reflected in guidelines to compose models described with the corresponding metamodels. This is an optional step because it is also possible to use workflow languages in isolation.

3. **Design & build applications.** In this phase applications tailored to specific languages or groups of languages are developed. These applications

can be used to build the models (editors), but also to interact with models at run time. This is also an optional step.

4. **Design & build workflows.** In the following phase, workflows to solve specific problems are designed and built. Each one of these workflows is an assembly, and thus it is formed by models, additional resources, composition information (a CCL program), and composition resources.

5. **Enact workflows.** The final step is to enact the workflows and interact with them at run time. This participation in the execution includes, for instance, providing data or taking decisions. Furthermore, the execution of these instances is monitored during this enactment phase.

Together with these groups of activities, we have also identified eleven kinds of participants, or roles, that are relevant to the whole process. These roles are the following:

- **Domain expert.** This role represents people that do not necessarily have technical knowledge, but have knowledge about the domain of the workflows. Domain experts are not only suited to contribute their domain knowledge to specific workflows, but they can also contribute to the design of languages.

- **Language developer.** This role corresponds to people in charge of designing languages, which are capable of capturing the knowledge of domain experts in suitable and expressive domain specific languages.

- **Metamodel developer.** This role represents people that have an in-depth knowledge of Cumbia and are able to design and develop metamodels for new languages. They are also capable of evolving existing metamodels in response to changes in the languages.

- **Engine developer.** This role represents the developers who complement the work of metamodel developers by developing the parts of the engines that are not supported directly by the Cumbia Kernel and the metamodel.

- **Metamodel tester.** This role represents people that are in charge of ensuring the quality of the metamodels implementation, and their conformance to the corresponding language definition.

- **Composition tester.** Similar to the metamodel testers, these people analyze the composition guidelines and try to uncover problems in them.

- **Application developer.** This role represents developers who build the additional required applications.

- **Model builder (analyst).** This role groups domain experts that participate in the design of specific workflows or models, by contributing their domain knowledge.

- **Model builder (developer).** This role groups technical people that also participate in the construction of specific workflows, for instance by developing the software to support additional requirements.

- **Workflow manager.** This role represents managers that are in charge of creating instances of the workflows, and monitoring their execution.
- **Participant.** Participants are the people that are part of the workflow execution and must perform steps that determine its execution or its outcomes. For instance, participants may have to interact with client applications to supply information or take decisions about the running workflow.

Between the list of roles just presented and the list of stakeholders proposed by Weske [Wes07] there are important differences. For instance, in the list of Weske there are two types of stakeholders that are responsible for the development of the "business process management systems" and the "software artefacts required to implement business process". In our case these roles are more detailed and form the core of the list. Conversely, Weske's list identifies six types of stakeholders that participate in the definition and enactment of the business process, whereas we only identified three roles that fulfill those tasks. The differences between these lists highlight the fact that our proposal is mostly targeted at the designers and developers of workflow languages and engines, and not so much at the users of those.

In the following sections we will explain in more detail every phase, the artifacts produced and required in each one, and also the tools available for each role.

Notation

In the following sections we use an ad hoc graphical notation, roughly based on BPMN's, to represent the different parts of the process. Figure 6.2 shows the structure of the diagrams and its main elements.

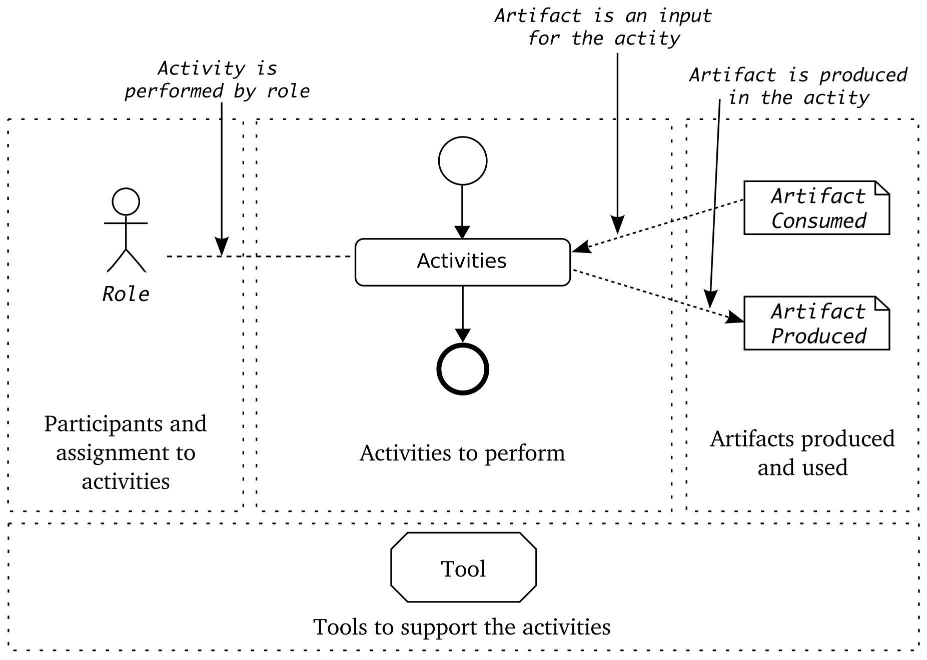

Artifact is an input for the actity
Activity is performed by role
Artifact is produced in the actity
Role
Activities
Artifact Consumed
Artifact Produced
Participants and assignment to activities
Activities to perform
Artifacts produced and used
Tool
Tools to support the activities

Figure 6.2 Graphical notation used to present the details about the process

In the central section of the diagrams (*Activities to perform*) we use BPMN's notation to show the relevant activities, and their order of execution. Besides tasks and flows, in some of the processes we use also BPMN's gateways and subprocesses.

The section to the left of the diagrams (*Participants and assignment to activities*) graphically represents the roles that participate in each of the activities of the central section. Dotted lines are used to connect each activity to the role that should perform it.

The section to the right of the diagrams (*Artifacts produced and used*) shows the artifacts that are relevant to the activities in the central section. Some of these artifacts are inputs to one or more activities, while others are produced by the activities. The difference between these artifacts can be seen in the direction of the arrow that connects them to activities. Finally, there are also some artifacts which are represented both as inputs and products of an activity: this means that these artifacts are updated by the activity.

The section of the diagrams located near its bottom (*Tools to support the activities*), depicts those tools that we have developed and can be used by the participants to perform some of the activities. We have not included any graphical element to connect these tools to specific activities, but the text makes explicit which tools support which activities.

6.2 Design and develop languages, metamodels and engines

The first phase of the process covers the steps to design and implement new workflow languages, their metamodels, and their engines. In this phase, the participation of Domain Experts is crucial: they provide the domain knowledge that languages and applications must incorporate. Furthermore, they define the requirements and functionalities for the applications. Domain Experts team up with Language Developers to specify the languages which are to be implemented and used in the subsequent steps. The work of Language Developers achieves an important objective: it captures domain knowledge in a suitable and expressive language. Finally, this phase also involves Developers which know the Cumbia platform and can use it to implement the support required to use the new languages.

Figure 6.3 presents a coarse view on the activities involved in this phase. Some of these were covered in chapter 4, when we showed how to implement a concern specific workflow language and an engine for it. In this section, we present these activities in the context of a development process, and we complement them with some additional steps. The final results of this phase are used in the subsequent phases and they are a language specification, a tested implementation of its metamodel, and an engine to run models.

In this phase of the process we also consider languages built as an evolution or a new version of an existing language. In the first parts of this section we mostly describe the activities to develop languages from scratch. Afterwards, in section 6.2.5, we describe how language evolution is incorporated in the process and how this alters the normal flow of activities.

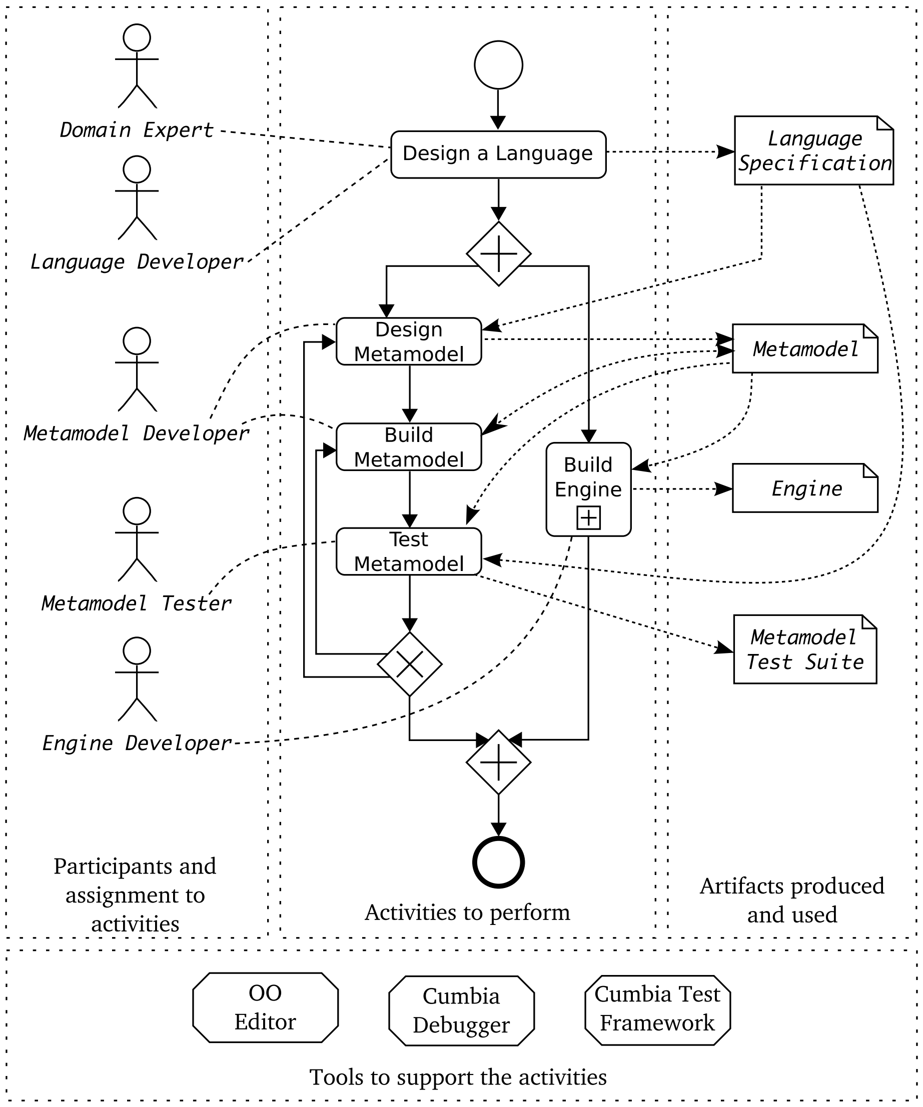


Figure 6.3 Design and implementation of a Cumbia based workflow language

6.2.1 Design a language

The first step in the process, to Design a Language, has two main goals. The first one is to design a language which captures the adequate amount of domain knowledge. For instance, it must capture the main entities of the domain, their relations, their interactions, and the relevant restrictions. The second goal is to design a language that domain experts can use effectively to address the problems they face. Depending on the domain, the specific requirement related to this may vary. For instance, in the domain of scientific based applications, users expect to have self explanatory workflows where it is easy to see if results make sense scientifically [MBZL09].

The most important input for this step is all the knowledge provided by domain experts. Besides contributing information about the domain, these users also define the requirements for the language and for the applications which will use the language.

Another input for this step are other related languages. Since Cumbia supports the integration of multiple concern specific workflow languages, the design of a new language must consider those other languages which may be related. This includes existing languages and languages under development. A pertinent example in the domain of business processes is the design of a new control flow language which is to be integrated with an existing language to describe time restrictions, and a language under development to describe data flow. By taking these other languages into account since this early phase, the new control flow language will better integrate with those other languages. A further consideration to take into account at this point is reusability: although the Cumbia platform provides features to make languages more reusable, the design of the language can also facilitate or obstruct its reusability.

The previous considerations are all relevant to CSWfLs. However, this does not mean that Cumbia cannot handle non-modularized workflow languages or even generic workflow languages, like BPMN or WS-BPEL. Therefore, the result of this step can be the specification of such a language. Additionally, this simplifies some of the ensuing phases, especially the second and fourth phase, since the interaction between languages does not have to be analyzed, supported, and taken into account when designing workflows.

The output expected from this step of the process is a Language Specification. This specification must describe the elements of the language, their structure, and their semantics. The syntax of the language is also likely to be designed in this step. However, since we are not considering the development of editors, this syntax is not relevant to the subsequent steps of our development process. On the other hand, the semantics of the language must be complete and its specification should not leave space for interpretation by metamodel developers. This can be achieved with any means available, but we are not going to automatically process this information in any way.

6.2.2 Metamodel design and implementation

Figure 6.3 shows that the following activities in the phase are to Design a Metamodel and Build a Metamodel. In reality, these are not simple, atomic activities. Instead, they are complex set of tasks that Metamodel Developers carry out to implement the Language Specification built in the previous step.

Figure 6.3 also shows that Build Engine is performed while the metamodel is designed, implemented, and tested. Since we are not decomposing Build Engine into more detailed phases, this concurrency is necessary to show the following considerations.

- Unless the engine is totally *generic*, its design depends on the metamodel and its implementation.
- The implementation of a metamodel can depend on elements of the engine or on services provided by the engine. In chapter 4 we referred to these services as Language Specific Elements. For example, in the case of a WS-BPEL implementation, the engine can provide a gateway to consume web-services for elements in the metamodel.
- The testing of some metamodels may be impossible to do without an engine, or require a complex mock infrastructure. Therefore some tests of the metamodels must be done using scenarios deployed into an engine.

This concurrency would be eliminated if we identified more detailed tasks, such as *define services that engine must provide*, *test isolated metamodel*, and *test metamodel and engine*.

Design metamodel

In this point of the process, the metamodel is designed. This begins by identifying which concepts of the language must be reified as entities in the metamodel. A one-to-one relation between concepts in the language and entities in the metamodel is not mandatory. However, the maintainability, the flexibility, and the extensibility of both language and metamodel can be favored by keeping concepts and entities as aligned as possible.

After metamodel entities are identified, they are structured. This means that the relations between entities are made explicit. Next, entities are given attributes, and then they are given behavior. The behavior defined for each entity must consider two aspects: on the one hand, the way in which instances of the entity must alter their internal state in response to external stimuli; on the other hand, the way in which instances of the entity should interact with other instances of the same entity or with instances of other entities in the metamodel.

Since these are metamodels to use in the Cumbia platform, the result of the previous steps must be the specification of a Cumbia metamodel. This includes the full descriptions of the open objects in it, together with the description of their state machines, the description of the interfaces of entities, and the description of the actions.

In order to support the activities in this stage, we have developed a tool called the `OO Editor` (Open Objects Editor). This tool is based on the Eclipse GMF platform, and it provides a graphical editor to define the entities and the structure of the metamodels. The `OO Editor` also provides the means to describe the internals of the open objects, including their attributes, the events they generate, the interface of the entities, and their state machines (see figure 6.4). The only aspect that the tool does not support is the specification of the behavior, that is, the implementation of the entities' methods and of the actions.

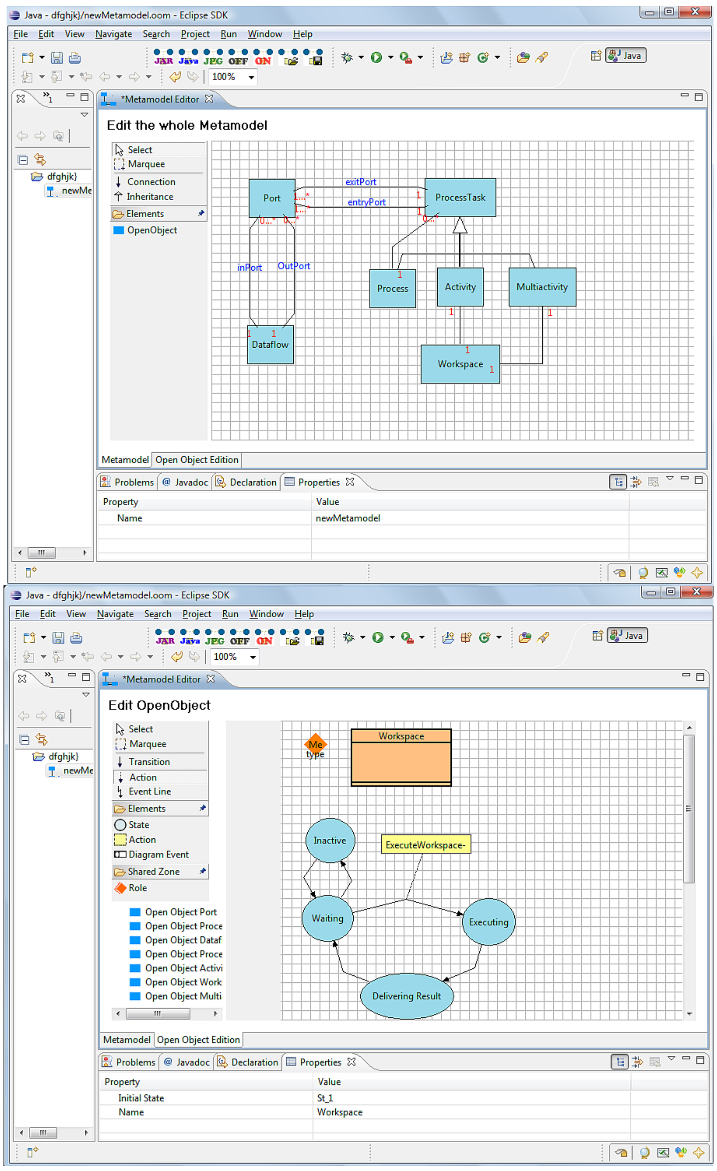
Given a metamodel specification, the `OO Editor` can export the XML files that are used to configure the Cumbia Kernel.

Build metamodel

After the metamodel has been designed, the following task is to implement it and prepare it to be tested or used in an engine. This involves writing the Java code for the entities and for the actions associated to state machines. Furthermore, additional files required in the metamodel have to be prepared (e.g. configuration files). Finally, the resources associated to the metamodel (the classes and the other files) are packaged in a *jar* file.

The `OO Editor` also provides support for some of the tasks in this stage. Given the specification of a metamodel, this tool can generate boilerplate code for the open objects' entities and actions. The generated files are full of markings that the Metamodel Developers must replace with the actual code for these classes.

In the end, the XML files exported from the `OO Editor` and the resources of the metamodel are tested and are used in the engine implemented for the language.



Property	Value
Name	newMetamodel

Property	Value
Initial State	St_1
Name	Workspace

Figure 6.4 The OO Editor editing a metamodel structure, and an open object

6.2.3 Metamodel testing

After the completion of a metamodel's implementation, this implementation must be tested[1]. The intention of this is to find errors in the design of the metamodel or in the implementation of the entities and the actions. Besides finding purely technical errors, this testing phase also aims to uncover inconsistencies between the specification of the language and its implementation in the metamodel. If an error is discovered in this phase there are two alternatives to correct it: the first one is to return to the Build Metamodel activity, fix the implementation, and then test again; the second one is to return to the Design Metamodel activity, fix the design, then fix the implementation, and then test again.

The proposed testing procedure is based on Model Based Testing [UPL06,

[1]In reality the metamodel implementation is not tested in isolation. As previously explained, in several cases this testing phase also involves the engine.

AD97] and on the usage of scenarios as figure 6.5 shows. Each scenario is composed of a model and of a set of stimuli for that model (e.g. the interaction of participants, or requests from external systems). The model is described using the language specified in the first step of the process. Since the semantics of the language is known, the behavior of the scenario, given the model and the stimuli, can be predicted[2]. On the other hand, the same model and stimuli are used with the metamodel implementation, and the actual behavior is recorded. The comparison of the expected behavior and the actual behavior can expose inconsistencies between the language and the metamodel.

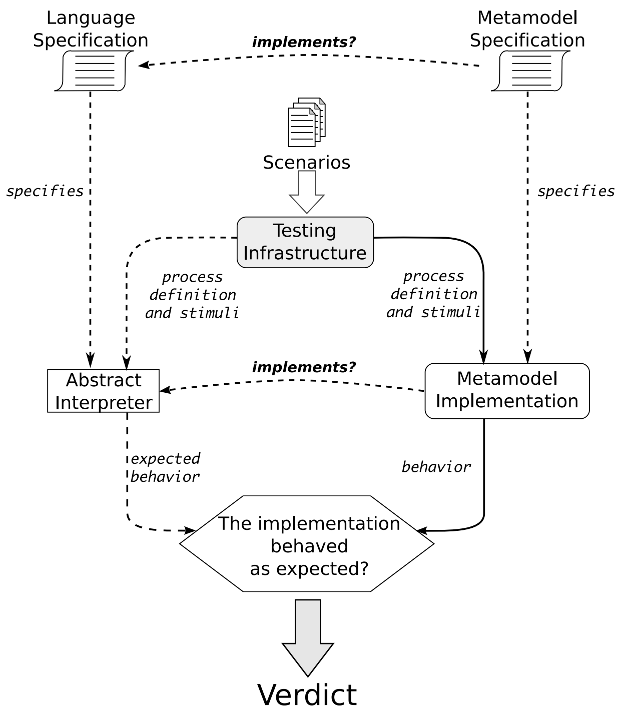
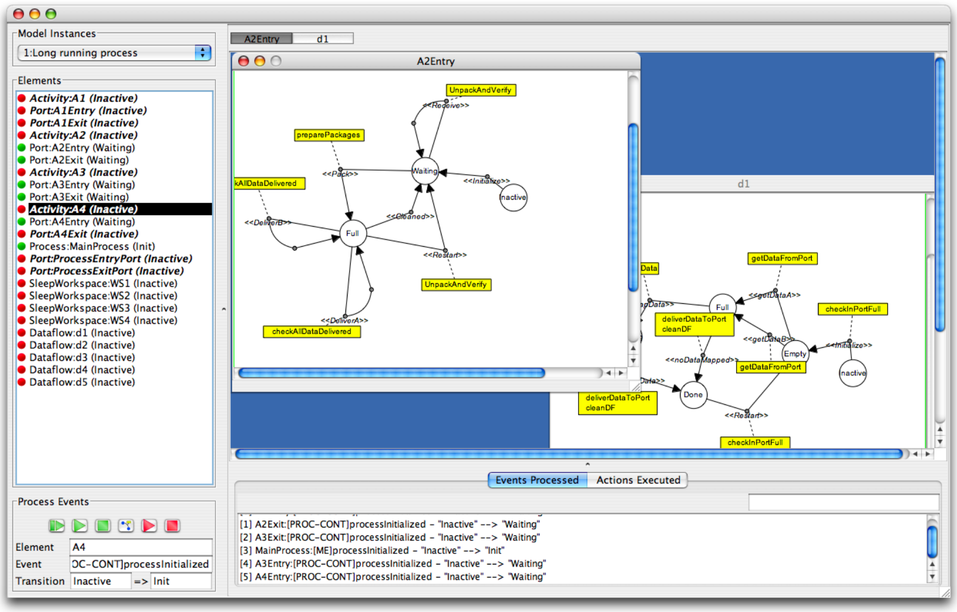


Figure 6.5 Scenario based testing of metamodels

It is important to note that it is not sufficient to compare the final results of the execution. Instead, it is necessary to check that the executed models reaches all the intermediate states and performs all the actions defined in the semantics of the language. For example, it may not be enough to just check the data produced at the end of a WS-BPEL process. Instead, it may be more appropriate to check that all the intermediate web-services were invoked with the correct parameters.

We have developed two different tools to support the work of Metamodel Testers. The first of those tools is called `Cumbia Debugger` (`CD`). It is an extension to the Cumbia Platform which provides a graphical interface that a Metamodel Tester can use to examine the execution of the models in an interactive way. With the `CD` they can see the structure of the state machines, the state changes, the actions executed, and the events generated and processed (see figure 6.6). The second tool available is called `Cumbia Test Framework` (`CTF`)[SJV10] and it is a collection of tools to build test workbenches and Test Suites for workflow languages. The `CTF` will be described in the next section.

[2]Unless there are existing implementations of the semantics, or there are formal models, the predictions have to be manually constructed. Therefore, the role of the *Abstract Interpreter* shown in 6.5 may be played by someone that knows the language and its semantics.



Activity:A1 (Inactive)
Port:A1Entry (Inactive)
Port:A1Exit (Inactive)
Activity:A2 (Inactive)
Port:A2Entry (Waiting)
Port:A2Exit (Waiting)
Activity:A3 (Inactive)
Port:A3Entry (Waiting)
Port:A3Exit (Waiting)
Activity:A4 (Inactive)
Port:A4Entry (Waiting)
Port:A4Exit (Inactive)
Process:MainProcess (Init)
Port:ProcessEntryPort (Inactive)
Port:ProcessExitPort (Inactive)
SleepWorkspace:WS1 (Inactive)
SleepWorkspace:WS2 (Inactive)
SleepWorkspace:WS3 (Inactive)
SleepWorkspace:WS4 (Inactive)
Dataflow:d1 (Inactive)
Dataflow:d2 (Inactive)
Dataflow:d3 (Inactive)
Dataflow:d4 (Inactive)
Dataflow:d5 (Inactive)

Element	A4
Event	OC-CONT]processInitialized
Transition	Inactive => Init

```
[1] A2Exit:[PROC-CONT]processInitialized – "Inactive" --> "Waiting"
[2] A3Exit:[PROC-CONT]processInitialized – "Inactive" --> "Waiting"
[3] MainProcess:[ME]processInitialized – "Inactive" --> "Init"
[4] A3Entry:[PROC-CONT]processInitialized – "Inactive" --> "Waiting"
[5] A4Entry:[PROC-CONT]processInitialized – "Inactive" --> "Waiting"
```

Figure 6.6 Screenshot of the Cumbia Debugger

The basic result of the testing activity is a Test Suite composed of a number of scenarios. If the `CTF` is used, then another result of this activity is a test workbench for the language. The Test Suite is not only useful when the language is first specified and implemented. It can also be used to perform regression tests when the metamodel is modified, or it can be the basis for a new test suite whenever the language is modified or extended. Finally, it must be considered that in all but a few cases, scenario based testing cannot guarantee the correction of the metamodel implementation with respect to the language specification. If no errors are detected, the confidence in the results only depends on the quality of the test suite.

Cumbia Test Framework

The `Cumbia Test Framework`[3](`CTF`) is a framework for creating test environments and test suites for metamodels [SJV10]. The `CTF` is not tied to any particular language. Instead, it defines the general structure of a scenario, and provides the base elements to define those scenarios, execute them, and verify if the execution proceeded as expected.

One of the main motivations behind the design of the `CTF` is to minimize the impact of concurrency in the result of the tests. Existing test frameworks, such as JUnit, offer very poor support to test concurrent programs [LHS03, Ric10]. Since concurrency is at the base of the Cumbia platform and the open objects, using a testing tool with issues to handle concurrency is not desirable. To counter this situation, the `CTF` relies on the off-line analysis of traces which are captured during the execution of the models. This approach was inspired by the work of Kortenkamp et al. [KMSF01].

[3]The development of the Cumbia Test Framework has been an ongoing effort, and several students of the Universidad de los Andes have contributed to its development. The contributors to `CTF` include Sergio Moreno, Camilo Jiménez, Carlos Vega, John Espitia, Ivan Barrero, Carlos Rodríguez, and Mario Sánchez.

In order to gather the information for the analysis, the CTF instruments the scenarios using *sensors*. These sensors are installed as actions in the open objects and they register run time information each time they are executed (i.e. the transition where they are installed is triggered). The information that sensors obtain is stored in traces. Afterwards, when the execution of the scenario is completed, those traces are analyzed to uncover situations where the execution did not follow the semantics of the language.

The elements in a CTF scenario are the same regardless of the metamodel under test. These elements are the following.

1. **Models.** These models are described using the metamodels under test and they provide concrete situations with known behavior. Scenarios can include a single model or several ones. For example, if a control-flow language allows the interaction between various processes, then the scenarios to test the language will have to include multiple models.

2. **Instantiation schemas.** The instantiation schemas define how to create the instances of the models in the scenario. For instance, the instantiation schema can determine that one instance of each model has to be created.

   The instantiation schemas also have an important role when the CTF is used for scalability testing. In those situations, the schema offers a practical way to request the creation of hundreds or thousands of instances of a model.

3. **Animation program.** The first part of a scenario (the models), provide only a static view on the scenario. The instantiation schema provides only a little bit of dynamic information, as it determines how the models should be instantiated. This dynamic information is complemented with animation programs that describe the stimuli that models require. For example, an animation program for WS-BPEL should define the responses of mock web-services to the requests made in the processes.

4. **Observation structure.** Each scenario requires different information to be gathered in order to check its correct execution. The observation structure for a scenario defines the types of sensors to use, which determine the information to gather in the traces. The observation structure also defines the placement of sensors in the models (i.e. the transitions where they should be installed as actions).

5. **Assertion program.** The last element of a scenario are the programs that check the information in the traces to discover inconsistencies between the expected behavior and the observed behavior. These programs can be built in two ways. The first one is to use a general purpose programming language (e.g Java) and an API to access the traces. The disadvantage of this is having to write Java code for each scenario. The second way is to use *data analyzers* and the *assertion language* defined by the CTF. Data analyzers are simple programs that look for specific patterns in the traces. Although they are also written in Java, they are easily reused across different scenarios. Furthermore, scenarios designers do not have to know the internals of these analyzers because they can use them through the *assertion language*.

Besides defining the elements that each test scenario must include, the CTF also provides tools to use those elements and perform the tests. All these elements and tools are generic, which means that they must be specialized for testing specific workflow languages. For instance, animation languages, to write the animation programs, have to be designed following the characteristics of each language under test. Three steps are required to adapt the `CTF` and use it to test a specific metamodel.

1. Specialize the Framework

The `CTF` is a framework to build testing environments for specific workflow languages. Therefore, in order to use the `CTF` and test a metamodel that implements a language, a new testing environment has to be developed. This is done by specializing three aspects of the `CTF`:

- *Animation Language.* The stimuli that the test framework provides to control the execution of the scenarios have strong dependencies towards the actual languages tested. Therefore, the animation languages to describe these stimuli have to be specialized for each tested language. In particular, each animation language can describe different characteristics of the stimuli, and can specify in different ways how and when these stimuli should be generated.

  For example, in the case of a testing environment for a WS-BPEL metamodel, the animation language should describe aspects of the mock web-services used in the scenarios. For example, it could describe the data contained in responses, or the delay to generate those responses.

- *Data Analyzers.* The data analyzers required for each tested language depend on the relevant assertions to test. Therefore, specific data analyzers have to be implemented.

  In the WS-BPEL example, interesting assertions could check the data produced by the processes, or the order of execution of web-service invocations. Therefore, a testing environment for WS-BPEL should include the data analyzers to find this kind of information in the traces.

- *Sensors.* Data analyzers depend on the data left in traces by sensors. Thus, specific sensors have to be created to record the data required.

  To fulfill the requirements of the data analyzers previously described, the sensors in a WS-BPEL testing environment should register in the traces the data produced by each activity, and the time of the execution of the invocations.

2. Build Scenarios

The second step in the usage of the `CTF` is to build the scenarios. This step can be divided in three main parts.

- In the first part, the requirements for the scenario are established. This means deciding which aspect of the language is going to be verified and which kind of assertions are going to be used.

- Afterwards, a scenario that presents the aspects to test has to be designed and implemented. This includes building the models involved, writing the animation programs, and establishing the instantiation schema.

- The final part involves both the observation structure and the assertion programs: first, the assertions that should be valid when the scenario is executed are established; then, the corresponding assertion programs are written; finally, an observation structure is defined to gather the information required in the assertion program.

Moreover, the construction of a scenario does not have to start from scratch each time, as many of their elements can be reused. For example, several scenarios can share most elements and only differ in their animation programs. Another common reuse situation is where the same models and animation programs are used to verify different assertions.

3. Execute the scenarios

At last, scenarios are executed and a response is produced: if given the information in the traces all the assertions are valid, then the scenario has a positive result; otherwise, a negative result is produced and the Metamodel Testers receive a report of the assertions that were violated.

Figure 6.7 shows the four distinct phases in the execution of a scenario in the `CTF`.

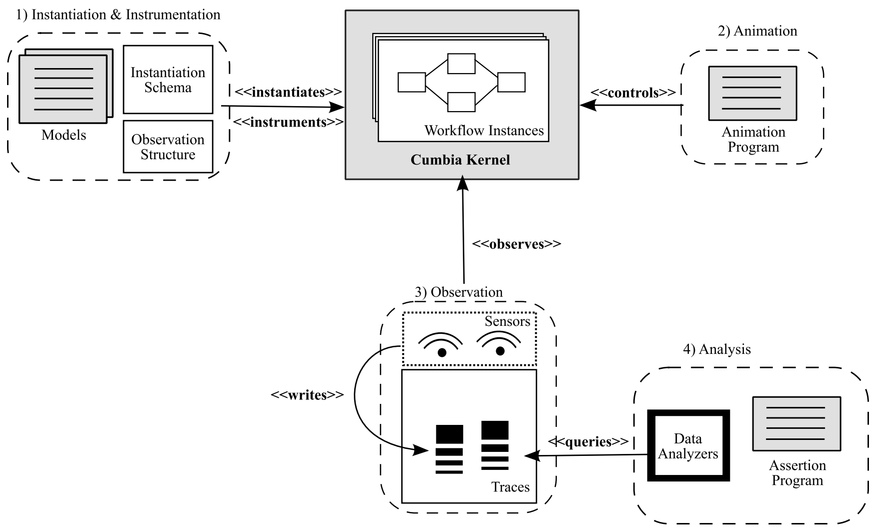


Figure 6.7 The phases of the execution of a scenario

1. *Instantiation and instrumentation.* In the first phase the models are instantiated according to the details found in the Instantiation Schema. Then, the model instances are instrumented as required by the Observation Structure. This means that sensors are installed as actions in the transitions of elements in the models. At last, model instances are further instrumented if the animation language requires so. For example, consider an animation language that has expressions of the form *"complete activity B after activity A has been completed"*. In such a case, the model instances will have to be instrumented with elements to notify the interpreter of the animation programs about the completion of activities.

2. *Animation.* In the second moment, the scenarios are executed. This is a responsibility of an interpreter of the animation language, which follows the instructions in the animation program.

3. *Observation.* While models are being executed, the sensors gather information and store it in traces: each time a transition with a sensor is triggered, the sensor is executed, retrieves runtime information, and stores it in a trace.

4. *Analysis.* In the final phase, after the execution of the models is complete, assertion programs are run. Using the information of the traces, these programs check that the execution of the models proceeded as expected.

6.2.4 Engine development

In chapter 4 we presented the structure of engines in Cumbia, and we showed how each of them incorporates an instance of the Cumbia Kernel. In this step, the goal is to design and build an engine to support and complement a specific metamodel. To design an engine the following aspects have to be taken into account:

- The additional services required by a metamodel (the Language Specific Elements). For instance, a metamodel may require gateways to communicate with web-services, or services to access a database.

- The additional interfaces that the language requires. For instance, the specification for WS-BPEL defines a standardized interface that all implementations of the specification must offer.

- Finally, the engines define how to handle and identify model specifications and model instances.

As we previously said, most of an engine's implementation is dependent on the metamodel. Therefore, the development of engines proceeds in parallel to the development of the metamodels. Furthermore, in figure 6.3 we have depicted Build Engine as a collapsed process because it must include a number of steps commonly found in the development of any software system (e.g analysis, design, implementation, and testing).

6.2.5 Language and metamodel evolution

Up to this point we have considered the activities necessary to design and implement a language from scratch. However, in many cases it is possible to develop a new language starting from an existing one. This is more likely to happen when the new language is an updated version of an existing language.

Figure 6.8 shows a more complete version of the activities that were previously described. According to this figure, the first thing to consider in the process is whether a new language is required, or if an existing one can be adapted. In the former case, the process proceeds as it was described in the preceding sections. In the latter case, the specification of the existing language is modified to reflect the new requirements, and it is stored with a new name or a new version number.

The following step is to determine if the metamodel of the existing language can accommodate the changes required in the new language. This is an analysis that will have to be done by hand in each case. Given the characteristics for flexibility and extensibility of the Cumbia platform, we expect that most changes will be attainable with a reasonable effort. However, in some cases it will be more

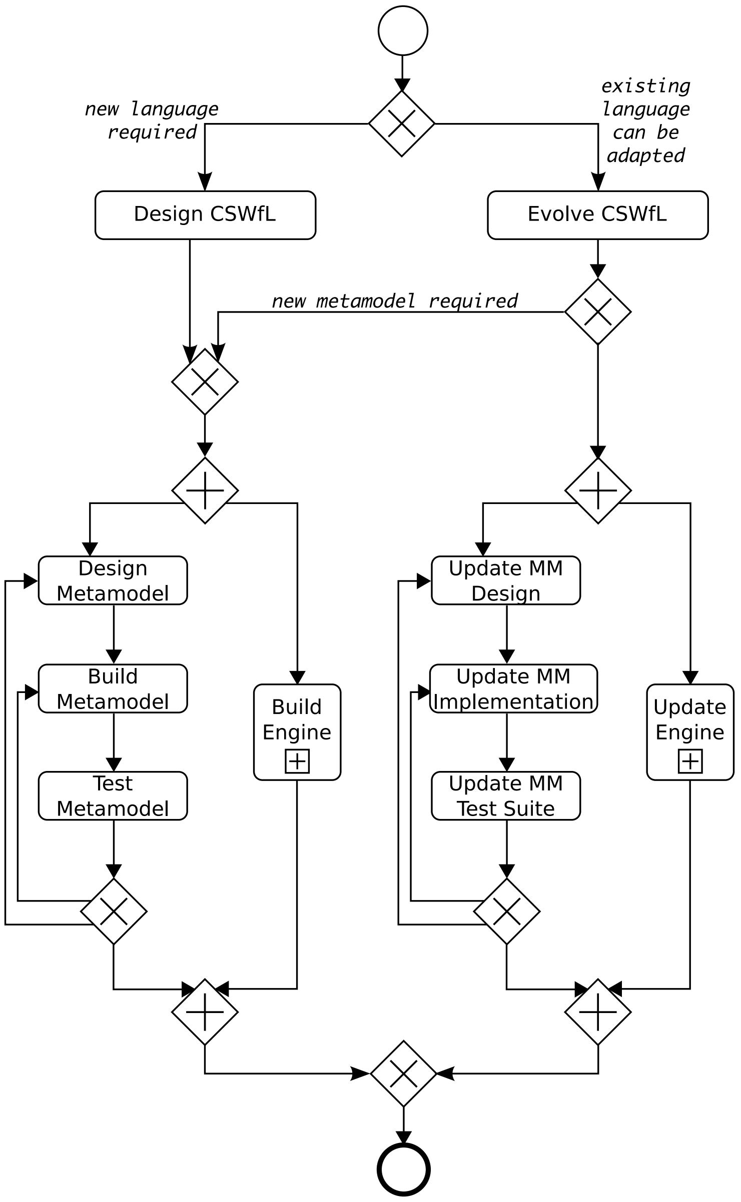


Figure 6.8 Steps to evolve an existing workflow language

convenient to design or develop a new metamodel. For example, the differences between BPMN version 1.1 and BPMN version 2 are so profound that it is not worth to convert one metamodel into the other.

When a metamodel is adapted, the rest of the activities are similar to the activities that were discussed in the previous sections. Furthermore, these activities result not only in an updated metamodel but also a updated test suite and updated engine. The investigation of whether it is more cost effective to develop a new language than adapting and extending an existing one are outside the scope of this dissertation.

6.3 Design metamodel composition

In the previous phase languages have been designed and implemented as metamodels. In many cases, these languages and metamodels are designed without explicit dependencies. Therefore, the goal of the next phase in the process is to establish how models described with these languages and metamodels can interact. This is achieved in four main steps. Firstly, a subset of the available languages that will be used together is selected. Secondly, the interactions between these languages are analyzed and documented. Then, this analysis is used to establish how models will have to be composed and coordinated. This is reported in guidelines to compose the selected metamodels. Finally, tests are done to verify that the composition guidelines do not create problems and inconsistencies in the models during execution.

Figure 6.9 shows the main activities of this phase, together with the participant roles and the artifacts that each activity uses or produces. In the following sections we discuss all these in detail.

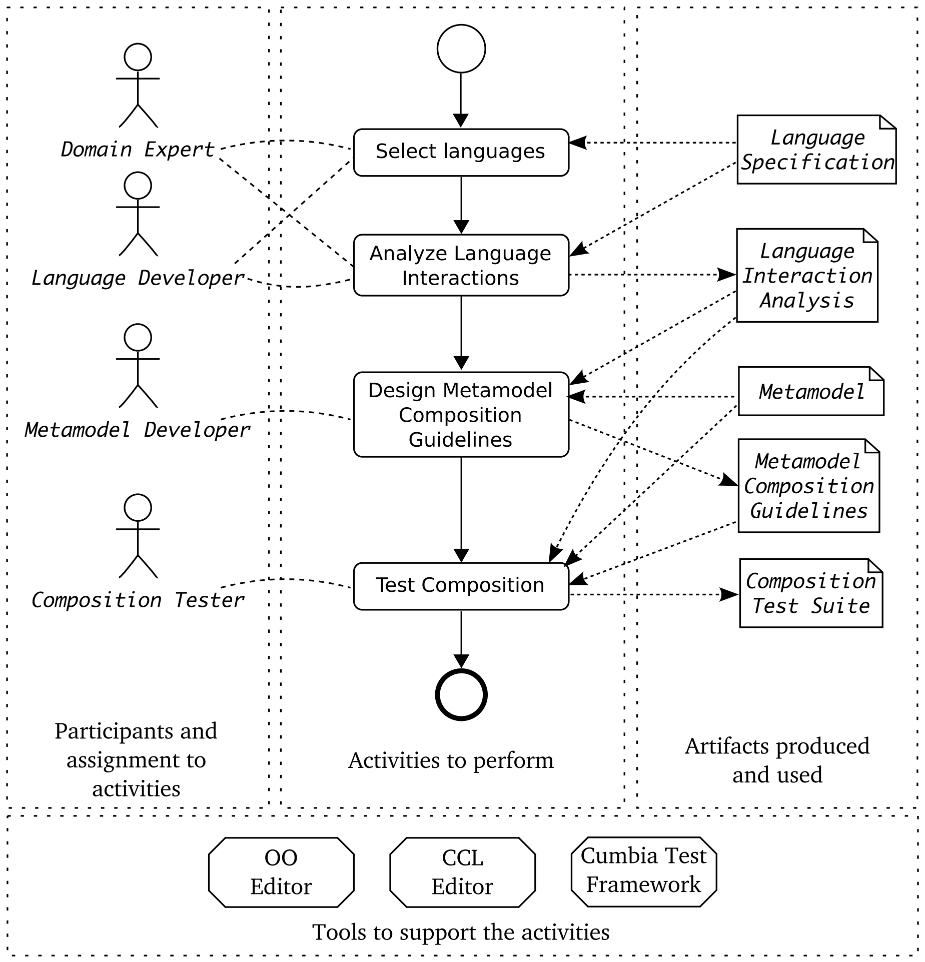

Domain Expert
Language Developer
Metamodel Developer
Composition Tester
Select languages
Analyze Language Interactions
Design Metamodel Composition Guidelines
Test Composition
Language Specification
Language Interaction Analysis
Metamodel
Metamodel Composition Guidelines
Composition Test Suite
Participants and assignment to activities
Activities to perform
Artifacts produced and used
OO Editor
CCL Editor
Cumbia Test Framework
Tools to support the activities

Figure 6.9 Details of the process to design the composition of metamodels

6.3.1 Select languages

The first step in this phase is to select a set of languages that will be used in some application. This selection depends on three factors: the general objectives of the application, the particular characteristics of the domain, and the compatibility of the languages. The first factor to consider are the objectives and requirements of the application. They determine the concerns that the application must handle, the information to manage, the interaction with external applications, and the actions that users can perform to control the processes. For example, an application's requirement can be to assign tasks based on the knowledge and skills of participants. This creates the need to handle that information about participants, and also necessitates a language to assign tasks based on that information.

The second factor to consider are the characteristics of the domain where the application is used. This is relevant because each domain determines important restrictions which can limit the applicability of a language. For example, some control flow patterns which are commonly not well supported in workflow languages, appear very frequently in certain domains. Therefore, it is important for the languages selected to work in that domain to support those patterns.

Finally, the selection of the languages to use in an application can also be influenced by how well the languages can be integrated: not every pair of concern specific workflow languages can be successfully composed. For example, some languages can be incompatible if they handle similar concepts in different ways that cannot be reconciled. Consider as an example a control flow language that handles multiple instances of the activities, and a language to define task assignments that cannot distinguish between different instances of the same activity.

The actors that participate in the activity Select Languages are of two kinds. On the one hand, there are Domain Experts. They are capable of establishing the requirements for the application and assess whether the selected languages fulfill these requirements. On the other hand, there are Language Developers, which have an in-depth knowledge of the languages available, and can judge whether the integration of the selected languages is feasible.

The result of this step is only a selection of the languages to use in a specific context. In the following step, the relations and interactions between those languages are thoroughly analyzed and then they are applied to the composition of the corresponding metamodels and models.

6.3.2 Analyze language interactions

The second step in this phase is to analyze the possible interactions between the selected languages. This is done by discovering how elements in the languages relate, and how these relations determine their behavior. In the case of the languages presented in chapter 5, the most easily seen interaction is between `MiniBPMNTasks` and `XTM` `Time Restrictions`. However, `Time Restriction` can also be related to `Processes` of `MiniBPMN`, and a single `Time Restriction` can be related to several elements of `MiniBPMN`(e.g. a task and a process).

Furthermore, the relations and interactions between language elements can also involve more than two elements. As an example, consider a language to express the workflow control flow (e.g. `MiniBPMN`), a language to express rules to offer and assign tasks to workflow participants, and a language to express time restrictions (e.g. `XTM`). In this context, a possible interaction between the three languages would be to have rules restricting the maximum amount of time a task can be offered after it becomes enabled to be executed.

As in the previous step, the only roles involved in this one are the Domain Experts and the Language Developers. At the end of the step, they have to report the relevant interactions between elements in the languages. This description should include at least the following aspects for each interaction pattern:

- The types, numbers, and characteristics of the elements involved. For instance, a `Time Restriction` that checks a time lapse in an `XTM` model can interact with two different `Tasks` in a `MiniBPMN`model. In order to differentiate them, these two `Tasks` could be characterized as the *initial* and the *final* tasks.
- The context in which the elements interact. In the previous example, the elements initiate their interaction when the *initial* task initiates its execution.
- The consequences of the interaction. In the example, after the *initial* tasks starts its execution, the `Time Restriction` should start checking whether the allowed time lapse is surpassed before the *final* task completes its execution.

The goal of this report is to have enough information about the patterns of interaction between the languages, to map the same patterns to the interaction between elements conformant to different metamodels.

6.3.3 Design metamodel composition guidelines

In this step, the patterns documented in the previous step are taken as the base to describe the corresponding interaction patterns between elements in the metamodels. In order to do this it is necessary to know how the elements of the language were implemented in the metamodel. For example, it is necessary to know how the high level events of the language correspond to events in the metamodel. Therefore, this step has to be performed by Metamodel Developers because they should have a deep knowledge of the metamodels and of their relation to the languages.

The result of this step is a document that describes Metamodel Composition Guidelines. This document has a structure similar to the document obtained in the previous step. However, in this case the description of the interaction patterns is done at the level of the metamodel and the open objects. This document thus provides the guidelines to compose and coordinate models conformant to the involved metamodels, including guidelines to write the CCL programs. These guidelines are provided in the way of patterns of CCL code where it is only necessary to replace the identifiers or paths of the specific elements that must interact.

At the time of the writing of this dissertation, we have a work in progress that has as objective to formalize the result of this step and facilitate its usage in subsequent phases[4]. In this work we are developing a language to first describe the interaction patterns between metamodels, and then facilitate the construction

[4]This work is being done with Carlos Rodríguez. Carlos is a current student of the Master in Engineering program at the Universidad de los Andes.

of the assemblies using high level languages based on those patterns. When this work will be completed, it would not longer be necessary to write CCL programs by hand. Instead, they will be generated from the high level descriptions. As a result, the complexity of building and assembly will be reduced, and it will become possible to ensure that the interaction between models follows the interaction patterns discovered.

6.3.4 Test metamodel composition

The goal of this step is to test the composition guidelines produced in the previous step. To do so, these guidelines are used to build test scenarios where models conformant to different metamodels interact. Then, the results of executing these scenarios are compared against the semantics of the languages and against the analysis of the interaction between the languages. Therefore, the Composition Test Suites produced in this step require as input the Metamodel Composition Guidelines and the report about the Language Interaction Analysis.

The tests to check the composition of languages are designed and executed in a similar way to the independent tests for languages. They are also done using the `Test Framework` and there are only three major differences. In the first place, the scenarios have to include CCL programs in addition to the model descriptions. In the second place, the languages to control and observe the models include additional elements to handle concurrent model instances in execution. Finally, the `Test Framework` has to interact with the engines through the `Cumbia Weaver` instead of doing it directly.

The testing tasks described in this step are performed by a Composition Tester. This role is fulfilled by users with an understanding of the composition mechanisms and of the `Test Framework`. To build the scenarios, they use the `OOEditor` (to create ad hoc metamodel extensions) and the `CCL Editor`. The final result of this step is a Test Suite that shows that the composition of the selected languages is correct when it is done following the composition guidelines established in the previous steps. This Test Suite can be later used to check that the evolution of the metamodels has not created problems in the interaction of the languages.

To simplify figure 6.9, we have removed some elements which show that when the execution of the test suite fails, the composition guidelines have to be corrected.

6.4 Design and build applications

All the metamodels and engines implemented on top of the Cumbia platform and the open objects share a number of common characteristics. This makes it possible to have generic tools to build, interact, control, and monitor the models and the assemblies. However, in many scenarios, generic applications are not adequate, and specialized ones have to be constructed. For example, in a corporate scenario it may be necessary to build a specialized monitoring tool to display information organized according to business goals. A further reason to build specialized applications is to integrate the concern specific languages from the user perspective.

Figure 6.10 shows a diagram for this phase of the process. This part is not very detailed because it is mostly an ad hoc process with a structure that

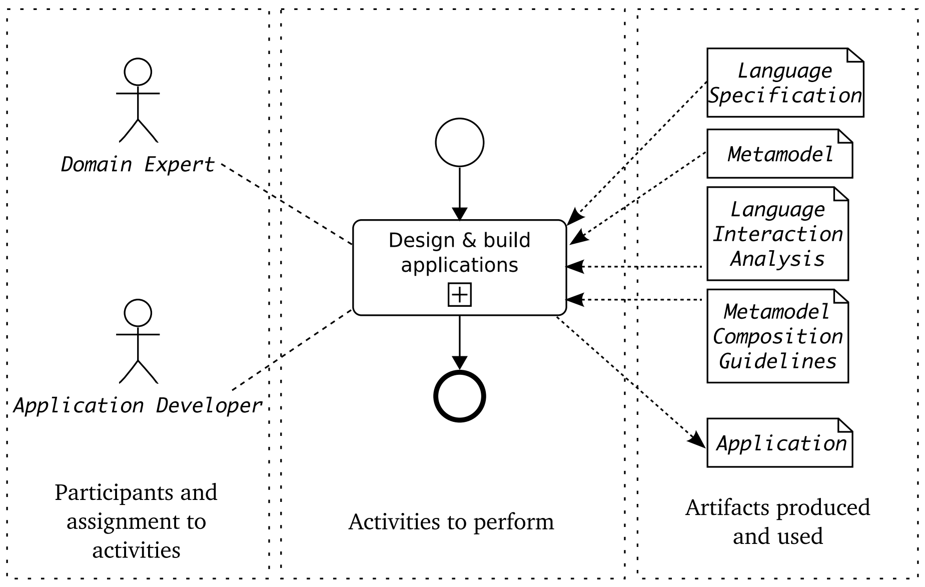

Domain Expert
Application Developer
Participants and assignment to activities
Design & build applications
Activities to perform
Language Specification
Metamodel
Language Interaction Analysis
Metamodel Composition Guidelines
Application
Artifacts produced and used

Figure 6.10 Details of the process to design and build applications

varies for every case. For instance, the activities to build a graphical editor that integrates all the languages used in a scenario, are very different from the activities to develop a web application that allows the participation of users in the workflows. Therefore, we did not attempt to describe the steps inside Design & Build Applications.

The relevant roles for this phase are the following: Domain Expert, and Application Developer. Domain Experts are relevant to this phase because they should establish the requirements for the applications, according to their specific needs. Therefore, in this group we have included not only the domain experts that participate in the design and specification of the languages, but also those that participate in the design of the workflows, and their enactment and monitoring. The other relevant role is that of Application Developers, which are responsible for designing and implementing the applications. In order to do so, they need a profound understanding of the languages and metamodels involved, and also of the Cumbia platform and the interfaces it offers.

There are four possible input artifacts for the activity of developing applications. On the one hand, there is the information about the languages, which is contained in the documents describing the languages (Language Specification), and in the corresponding Metamodels. The Metamodels also provide implementation information such as the interfaces of the elements, and their relations. Finally, the other inputs are the Language Interaction Analysis and the Metamodel Composition Guidelines, which describe how applications should handle the interaction between languages and, correspondingly, how they should deal with metamodel composition. By providing these artifacts produced in the previous phases, this phase can produce applications, and they can be consistent with the structure of the languages and with their composition.

In summary, the phase of designing and developing applications is where specific applications are built to complement generic applications. The details about this phase can vary depending on the kind of applications built, and even the whole phase can be skipped if no specific applications are required.

6.5 Design and build workflows

This phase of the process uses what was developed in previous phases to develop workflows that solve specific domain problems. This phase includes both high level activities, which do not directly involve the Cumbia platform, and low level activities which create Cumbia artifacts. Because of this duplicity, this phase involves two roles. The first role, Model Builder (analyst), is fulfilled by those that contribute their domain knowledge to design the workflows using concern specific workflow languages. The second role, Model Builder (developer), is fulfilled by those that have a technical knowledge of Cumbia and the metamodels, and are therefore able to implement the workflows designed by the analysts. The final result of this phase is a *cumbiar* file which can be loaded in the Cumbia platform.

Figure 6.11 shows the main activities, artifacts, participants, and tools that are relevant to this phase of the process. These elements are described in more detail in the ensuing sections.

6.5.1 Design a workflow

The first part of the phase starts with a very high level activity, Analyze the Workflow. In this activity, the objectives of the new workflow are established and then its general structure is designed. The goal of this activity is not to have a complete understanding of the workflow structure, but to have a high level view to later guide the design of the concern specific models.

After this initial activity, the more formal tasks of designing the models and their interactions are carried out by the Analysts. In this part of the process, the Analysts should use concern specific workflow languages to describe the various parts of the workflow. This results in a collection of Concern Descriptions. Roughly at the same time, the analysts must also produce a description of the intended interactions between the concerns of the workflow. These interactions should be based on the interaction patterns between the languages discovered in the second phase of the process (see section 6.3.2).

Because of the scope of the work presented in this dissertation, which focuses on the execution aspect of the workflow languages, we do not currently offer tools to support this part of the process. Therefore, analysts have to use ad hoc editors for the CSWfLs. We consider that future work related to this dissertation should look for approaches to easily build or generate editors for these languages, and make them compatible with Cumbia.

6.5.2 Build models, extensions, and resources

In the second part of the phase, the high level workflow is converted into concrete model definitions conformant to Cumbia metamodels. Since specific workflows usually require specialized elements, this part of the phase also includes the creation of metamodel extensions and their associate resources. In figure 6.11 we have identified two coarse activities that are executed in parallel, namely Build Models and Develop Extensions and Resources.

The objective of the activity Build Models is to transform the Concern Descriptions into Model Definitions that can be loaded in a Cumbia engine. Ideally, the editors used to write the descriptions should be capable of exporting the high level descriptions into the XML schema that Cumbia uses to specify the struc-

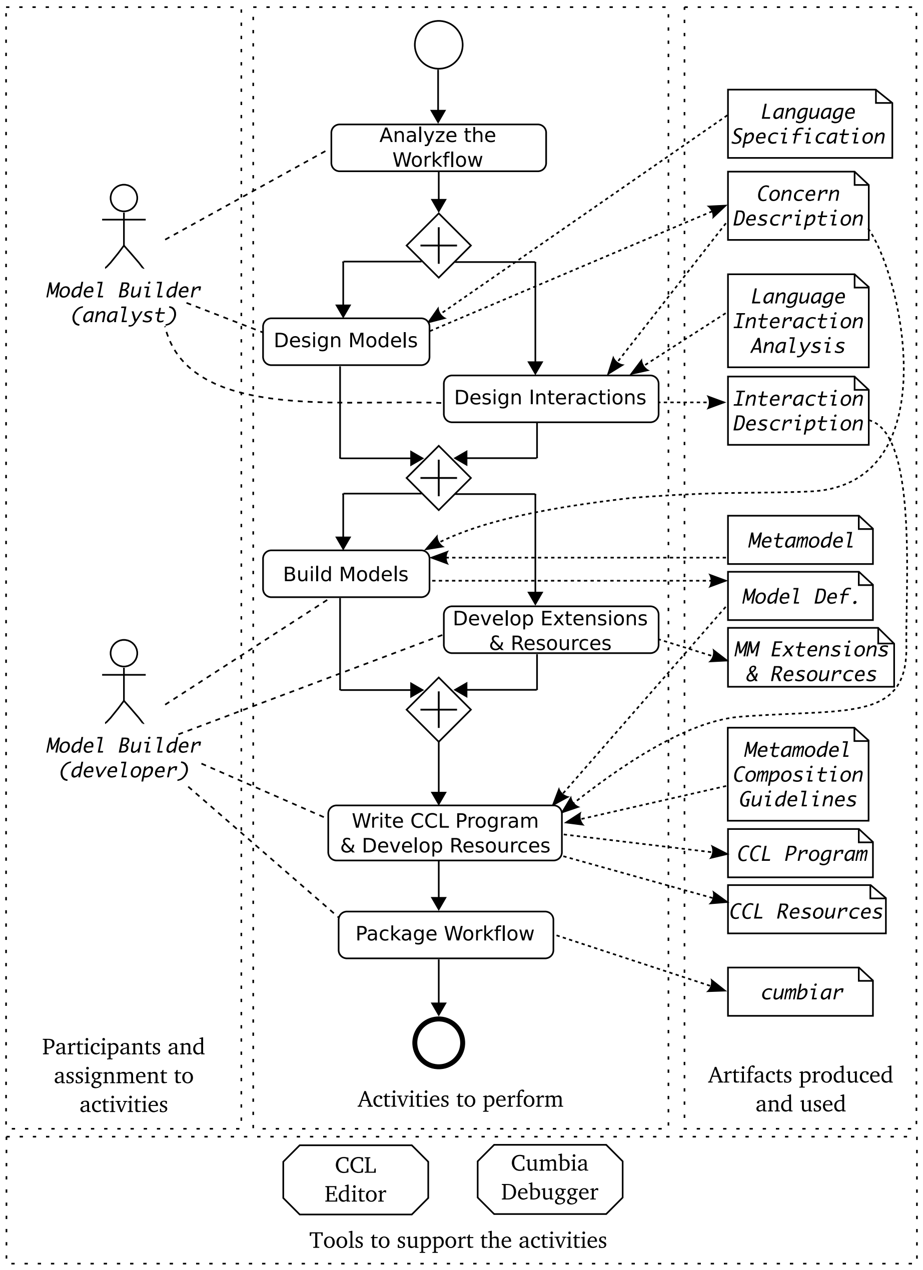

Analyze the Workflow
Model Builder (analyst)
Design Models
Design Interactions
Build Models
Develop Extensions & Resources
Model Builder (developer)
Write CCL Program & Develop Resources
Package Workflow
Language Specification
Concern Description
Language Interaction Analysis
Interaction Description
Metamodel
Model Def.
MM Extensions & Resources
Metamodel Composition Guidelines
CCL Program
CCL Resources
cumbiar
Participants and assignment to activities
Activities to perform
Artifacts produced and used
CCL Editor
Cumbia Debugger
Tools to support the activities

Figure 6.11 Details of the process to design and build workflows

ture of models. However, given the current lack of editors for the CSWfLs, this conversion has to be done using ad hoc converters implemented especially for each language.

On the other hand, the objective of the activity Develop Extensions and Resources is to build the elements that metamodels cannot provide because they are specific to the workflow. Among these additional elements, the most common ones are extensions to the metamodels that specialize the behavior of some open objects (see chapter 4). Besides the definition of the extended metamodel, these extensions also require the inclusion of additional classes where the specialized behavior is implemented (entities and actions). On top of the elements to support the extensions, the resources associated to a model can also include descriptors, configuration files, documents, or any other kind of file. For example, in the case of a model representing a WS-BPEL process, the associated resources must include the files that describe the *partner links* of the process.

The final result of this part of the phase is a set of Model Definitions complemented by Metamodel Extensions and Resources. At this point, these elements can already be loaded in the corresponding engines to be executed. However, unless their executions are correctly coordinated, they will not provide the semantics of the complete workflow. Therefore, the final steps of this phase are used to develop the elements to achieve the coordination of the models. Furhtermore, all these elements are packaged to manage them as single units.

6.5.3 Write CCL programs and package the assemblies

Finally, the last part of this phase takes the models built previously, assembles them, and produces a `cumbiar` file that can be loaded in the `Cumbia Weaver`. This is done in two steps. In the first one, Model Builders write the CCL Program to relate the models and coordinate their execution. The base for this program is the definition of the models and the description of their intended interaction. Furthermore, the CCL program should follow the Metamodel Composition Guidelines. In some cases, this step also includes the development of additional resources for the CCL program, i.e. ad hoc actions required for action based relations (see section 5.3.2).

The tasks related to writing the CCL programs and developing the resources are performed by Model Builders which currently can use the `CCL Editor`. However, this tool currently only provides very limited assistance. In particular, the `CCL Editor` cannot guarantee that a CCL program respects the composition guidelines. The ongoing work that we mentioned in section 6.3.3 is intended to solve this by providing better language and tool support.

In the second and final part of this phase, all the artifacts relevant to the workflow are packaged to obtain a `cumbiar` file. To avoid cluttering the figure, we have not made explicit the inputs for this activity in figure 6.11 but we now enumerate them:

1. The Model Definitions - these are XML files.
2. The Metamodel Extensions and the corresponding Resources - these are XML files and *jar* files with the additional classes and resources (e.g. configuration files, descriptors, images, etc.).
3. The CCL Program - this is only a CCL file.
4. The CCL Resources - these are *jar* files with the additional classes and resources (e.g. configuration files, descriptors, etc.).

6.6 Enact workflows

The final phase of the process described in this chapter is to enact the workflows built in the previous phases. The fine details about some of the activities in this phase were already discussed in section 5.3.2. In particular, in that section we described the process to load and instantiate a workflow packaged in a `cumbiar` file.

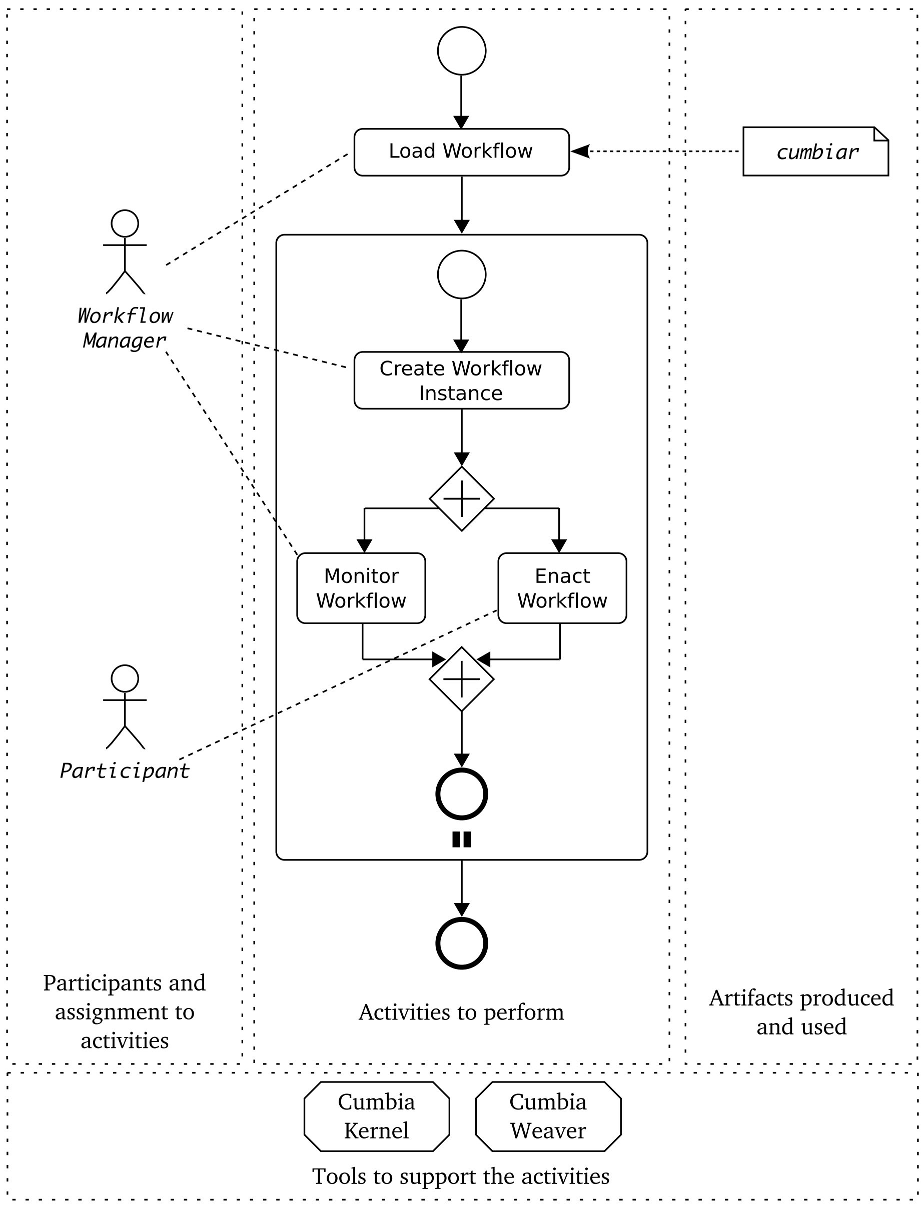


Figure 6.12 Details of the process to enact workflows

Figure 6.12 shows the main activities in this phase. In the first place, the `cumbiar` obtained in the previous phase of the process is loaded in the `Cumbia Weaver`. By doing this, the workflow becomes available to be instantiated and executed. The following step is precisely to create an instance of the workflow assembly. During this step, the CCL program in the assembly is executed and the necessary model instances are created in the corresponding engines. These two steps are performed by users fulfilling the role of Workflow Managers. The figure also shows that after a workflow is loaded, it can be instantiated and executed multiple times.

After the step where the workflow is instantiated, the obtained instance can be executed. This execution involves three elements. On the one hand, there are the users that participate in the workflow execution through client applications. These are the Participants in the workflow, and they are usually knowledgeable about the domain. The second element are external systems that interact with the workflow, either in an active or reactive way. Finally, there are the engines themselves that update the state of the models and execute actions in response to conditions, to the interaction of the Participants, and to the stimuli provided by the external systems. Moreover, there are domains where the workflows are completely automated and self-contained. Therefore, these workflows do not rely on the interaction with users or with external systems.

At the same time, the execution of workflows can also be monitored by the Workflow Managers. By doing so, these managers obtain operational information about the execution, such as the number of active instances or the mean duration of their execution. Furthermore, these managers can also gather run time data about domain specific aspects of the models. For instance, in the Fast Credit Process example, Workflow Managers can monitor the number of credit requests accepted and rejected. These monitoring activities are achieved with specialized or with generic tools. The former group contains monitoring applications built for specific domains and specific sets of languages. The latter group contains monitoring applications that can be configured and adapted to be used with different sets of languages.

6.7 Summary

In previous chapters, we presented the open objects' framework, CCL, the Cumbia Kernel, the Cumbia Weaver, and all the other elements that form the core of our proposal to support the development of workflow engines. In those chapters, the focus was on the characteristics of these elements and on the reasons for proposing their usage. However, we left out of those chapters the practical view on how these elements should be used, and on the ways to manage the complexity that they introduce.

To resolve this, in chapter 6 we have focused on the usage of the elements in the Cumbia platform by structuring one possible process to develop workflow engines. For this, we identified the main activities to perform, and how each one relates to different elements of Cumbia. We described the relations between these activities using a BPMN process where we made their dependencies explicit. To achieve this, we relied on the inputs that each activity requires, and the outputs that each activity produces. To complement this description, which only amounts for the control and data perspectives, we characterized the roles that people developing workflow engines with Cumbia should assume. Finally, in this chapter we also described additional tools that we have developed to facilitate the usage of Cumbia, and that workflow engines' developers should use. We consider these tools to be outside of the core of the proposal, but they have an important function in reducing its complexity.

7 Validation

Software design is not an exact science and it is not possible to prove (in the mathematical sense) that a proposed set of tools, architectural blueprints, and methodologies will serve to build any application in a specific domain. Instead, case studies can be used to illustrate the capabilities of a proposal and to do a qualitative analysis of its advantages and disadvantages. A crucial step in this approach remains in the selection of the case studies, because meaningful results can only be obtained if the case studies include the key aspects of the domain under study. Applying a validation approach based on case studies to Cumbia not only enables a critical analysis of the proposal, but also serves to discover and document recurrent solutions to key problems in the workflow domain.

In previous chapters, and especially in chapter 2, we have characterized the applications that we aim to support with Cumbia. We have also illustrated the approach by discussing the implementation of two concern specific workflow languages (`MiniBPMN` and `XTM`). In this chapter, we present other case studies which explore other relevant aspects in the workflow domain. By means of these scenarios, we provide more evidence about the benefits and disadvantages of Cumbia, and we examine how the objectives stated in the introduction to this dissertation were fulfilled.

The chapter starts by revisiting said objectives and then nine case studies, each one based in a different workflow language, are described. For these case studies, we first present a general description of the language, and then we describe the most relevant aspects of its implementation, and of the experiments performed (e.g. extensions, additional requirements, tool integration). Finally, we reflect about each scenario and on the features of Cumbia that it highlights. The final part of the chapter reviews the case studies, and analyzes them from the perspective of the objectives and characteristics presented in chapter 1.

7.1 Objective revisited

The general objective of this dissertation is to support the development of flexible engines for new workflow languages. To achieve this, we proposed an alternative to the current frameworks and architectures for building engines, and we implemented this proposal in a platform called Cumbia. In the rest of the chapter we use a number of case studies to show how the general objective is achieved with this platform.

In the introduction to this dissertation we claimed that four characteristics of the platform were central to achieve the objective. We will now revisit these characteristics and explicitly show how they appear in Cumbia. Afterwards, in each case study we show how the characteristics were stressed and their contribution to achieving the general objective.

C1. The platform is independent from particular workflow languages and execution models

The general objective of the dissertation implies that our platform should support a large number of existing and future workflow languages, including generic and domain specific languages. Given the restrictions found in platforms based on intermediate languages, a key characteristic for our platform was to be independent from particular languages or models. To achieve this, for designing the platform we used structures and abstractions that were powerful enough to be used in a large number of workflow languages, and which introduced the least possible number of restrictions. These elements are the open objects and the metamodeling platform itself, and their characteristics have been described in previous chapters. In this chapter, the case studies show how these elements were used to implement languages with very different characteristics. Furthermore, for some languages we describe how some of their characteristics were difficult to implement using open objects and how we overcame these difficulties.

C2. The platform maintains a clear mapping between language elements and implementation elements

This characteristic has two aspects to consider. The first one is that languages and their implementations should be aligned. This means that the mapping between elements of the language and elements of the engine implementation should be easy to discover and to follow. The second aspect is that languages, engines, and mappings should be easy to modify and keep up-to-date. In Cumbia this characteristic is supported by the usage of metamodels based on the abstract syntax of the languages. By avoiding a code generation strategy and instead implementing the semantics of languages as part of the metamodels, it becomes easy to trace the executable semantics of languages into the implementation. Furthermore, since changing the languages requires changing the metamodels, the mapping between the two elements is automatically kept up to date.

C3. The platform supports language flexibility

Because of the general objective, an important characteristic for the platform was to support language flexibility. There are three aspects of Cumbia that are related to supporting this characteristic. In the first place, there are the elements described for characteristic **C2**, because they facilitate the understanding of languages' implementation and thus favor their maintenance and evolution. In the second place, there are the extensibility mechanisms of the open objects, which define a taxonomy of easy-to-apply modifications at the element level. These modifications can be composed to form complex language level extensions and adaptations. Finally, the third aspect of Cumbia that contributes to language flexibility is the support for loosely coupled concern specific workflow languages. These languages are easy to add, remove, or replace in existing applications. Although this is not the same as language flexibility, it achieves similar results. Besides showing how languages were implemented, this chapter also presents a number of experiments where languages were modified and where CSWfLs were added.

C4. The platform is reusable and supports the implementation of open engines

Throughout this dissertation we have shown that the Cumbia platform offers many elements to reduce the effort required to implement a workflow engine and support a new language. These elements include the metamodeling platform, the open objects framework, the Cumbia Kernel, the Cumbia Weaver, the Cumbia Test Framework, and the Cumbia Debugger. All of them were designed deliberately to be language independent, but they can be specialized for specific languages. They can be reused for each new language that has to be supported.

Besides reducing the effort required to implement a new engine, Cumbia also presents some characteristics that facilitate the construction of open engines. The first one is offering very open interfaces that can be used to control and query most aspects of a workflow execution. Secondly, there is the usage of executable models, which contributes to aligning languages and their implementations. And thirdly, there is the widespread usage of events which can also be captured by applications external to the engine itself. By supporting open engines, the construction of the tool chains required for each engine is facilitated. Furthermore, since the base platform is shared by the engines of many languages, the applications that interact with these engines can use the same interfaces, thus opening the possibility of reuse. Some examples of this that we discussed in previous chapters include the Cumbia Debugger and the Cumbia Test Framework. The case studies described in this chapter present some further examples.

7.2 Overview of the case studies

To investigate how suited was Cumbia to implement flexible engines for workflow languages we did a number of case studies. These case studies first involved developing engines to test the expressive power of the platform, and then we tested their flexibility by adapting and extending the languages. The case studies were not all implemented on top of the same version of Cumbia. Instead, they were built progressively, and thus they served to guide the development of the platform by identifying the characteristics that it lacked.

The selection of the languages for the case studies was made based both on some technical characteristics (e.g. the type of the underlying execution model) and on their popularity. Furthermore, some of the languages selected are generic, while others target particular domains. Figure 7.1 shows, in a graphical way, how the languages selected can be distributed across the spectrum of workflow languages. We do not claim that we covered the whole spectrum, but we believe that we addressed an important part of it.

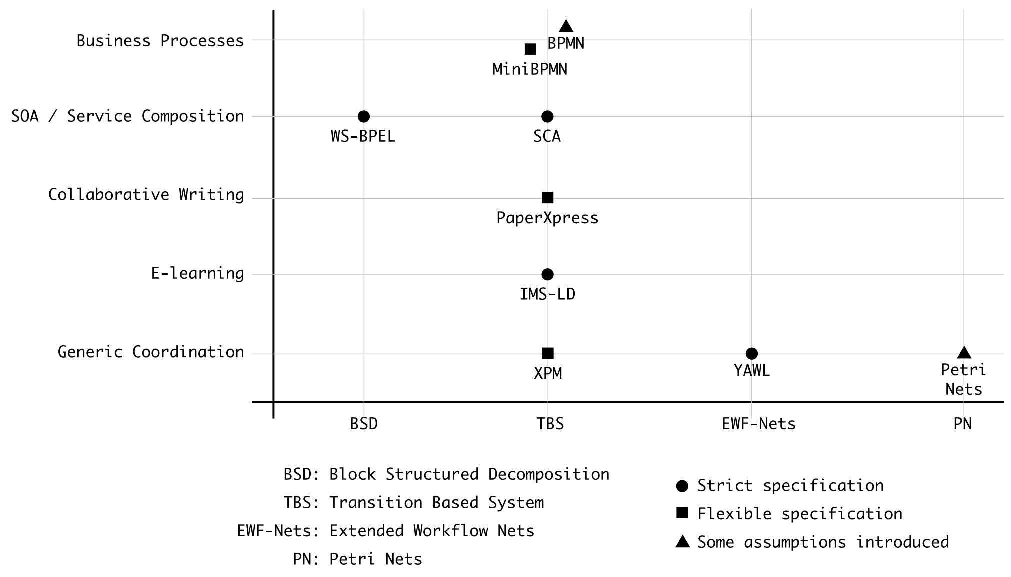


Figure 7.1 Characteristics of the case studies

In the figure, we have used the x axis to classify the languages according to the type of their underlying execution models. Most of the languages tested can be classified as *transition based systems* (TBS). This is normal considering that most workflow specification languages are based on this type of model. With respect to the types of execution models described in chapter 2, the only one not represented in these case studies is *rule based execution* (RBE), which is too different from Cumbia to be supported.

The y axis of the figure classifies the languages according to the domain that they target. In this graph, we have put WS-BPEL in the SOA / Service Composition domain, because the language was supposed to target that domain in its beginning; given the widespread usage of the language, we can also put WS-BPEL in the Generic Coordination domain. Similarly, we put BPMN in the Business Processes domain because that is its explicit target domain, although it can also be used for Generic Coordination.

Finally, the graph has a third dimension related to how close to the language specification was our implementation. There are three categories in this dimension. *Strict specification* refers to case studies where the languages were fixed and their semantics were clear. In those cases we implemented the semantics without making assumptions, although in some cases we did not implemented them in full. *Flexible specification* refers to case studies where we were able to change the specifications to suit our needs because we were the ones creating them. *Some assumptions introduced* refers to case studies where we implemented existing models or specifications, but where we had to made assumptions about their semantics. In the case of BPMN, we had to make these assumptions because the BPMN specification does not offer complete and unambiguous semantics. In the case of Petri nets, we had to make these assumptions to introduce transitions with non-zero durations.

The number of languages implemented in these case studies is relatively large and each one required a big effort to be completed. For each one it was necessary to study or design the language, and then to implement it in the Cumbia platform using metamodels and open objects. To test these implementations, testing environments were implemented using the CTF and then test suites with meaningful test scenarios were designed and implemented. For some languages, additional applications such as web-based clients had to be designed as well. Furthermore, some case studies involved extensions and adaptations to the languages and their tools. Several students at the Universidad de los Andes, which were doing their Master's thesis under the guidance of Jorge Villalobos,, had a central role in developing these case studies. Without their hard work, the scenarios available to test and analyze Cumbia's capabilities would have been fewer and possibly less useful.

In the following sections we will describe the case studies and present the most relevant details of their implementations. We have divided these case studies in two groups. The first one contains those that were more challenging to implement in Cumbia and which provided the most interesting information about the capabilities of the platform. The second group contains the rest of the case studies. For the former, the descriptions in the following sections will be quite detailed; for the latter, the descriptions will be more superficial because the implementation details to report are less interesting or are similar to those found in one of the languages of the first group.

7.3 `MiniBPMN`

`MiniBPMN` is an ad hoc, high level workflow language that we designed as a scenario for Cumbia. It is roughly based on BPMN, and thus its underlying execution model is based on a transition system. Since previous chapters already discussed the design and implementation of `MiniBPMN`, we are not going to discuss it any further in this chapter. In particular, chapter 4 described the language and presented in detail its implementation, its metamodel, and its engine. This served to show how a workflow specification language can be implemented on top of open objects and the Cumbia Kernel. That chapter also presented three extensions to the language, which illustrated the mechanisms in Cumbia to support language extensions at the metamodel level.

Chapter 5 continued the exposition of this case study by showing how the control concern was complemented with the time concern. We used `XTM` as the metamodel to describe time, and we showed how CCL was used to describe the relations between `MiniBPMN` models and `XTM` models.

In spite of its simplicity, the `MiniBPMN` scenario has enough elements to illustrate the main objective of the Cumbia platform and its main characteristics. On the one hand, we were able to implement an engine for the language without depending on a specific workflow execution model or an existing workflow language (**C1**). Furthermore, the implementation was flexible and thus we were able to change the language in several ways (**C3**). All the changes to the languages were implemented by changing the metamodel, or by composing a new metamodel (`XTM`), and thus we managed to kept a straightforward mapping between the language and the implementation (**C2**). Finally, the engine was implemented on top of a platform that provided many of the functionalities commonly required by workflows, and thus the implementation of the language was greatly simplified (**C4**). For example, the implementation of the `MiniBPMN` engine addressed concurrency issues only through the structure of the state machines, and it did not have to consider the management of threads and their synchronization. As a further example, the `MiniBPMN` engine used the persistency mechanisms offered in the platform by default.

7.4 YAWL and Petri nets

General description

The goal of this section is to present a case study where we implemented engines for YAWL and for Petri Nets. Our initial objective was only to support YAWL, but the close relation between Extended Workflow Nets (YAWL's execution model) and Petri nets, led us to implementing an engine also for the latter[1]. With this second engine we were able to evaluate some of the most complicated aspects of YAWL, which have to do with concurrency and synchronization.

YAWL (Yet Another Workflow Language) [vdAtH06] is a formal workflow language which originated in the academic community but has also been used in commercial applications. We already introduced YAWL and discussed its most important characteristics in chapter 2. In this chapter we provide some extra details about its structure and semantics.

We chose YAWL as a case study for Cumbia because of three main reasons: YAWL's fame in the workflow community; YAWL's precise semantics based on atomic state changes; and its simplicity with respect to external bindings.

In the first place, YAWL is well known in the workflow community. Its notation and semantics are common place, and it subsumes the core elements found in most workflow languages. Furthermore, YAWL is regarded as been very expressive, as it supports most of the control flow patterns. On top of that, YAWL has been frequently used as a case study in workflow research: many innovations in the field have been illustrated using YAWL, or have been compared to YAWL's own strategies.

The second reason for selecting YAWL is related to its precise semantics. Unlike languages where the semantics are informally defined, such as BPMN, in YAWL the semantics is formally specified using Extended Workflow Nets (EWF-Nets). EWF-Nets are themselves formally defined in terms of Petri nets. Because of this formality, implementing the language does not require a subjective interpretation of its specification. In addition to this, YAWL semantics have a characteristic that differs from several other workflow languages. In YAWL, the state of an entire process is updated in an atomic way, just as Petri nets' markings are updated in discrete steps. Also, the state changes of each element depends not only on its state, but also on the state and behavior of other elements. Since Cumbia is mostly based on concurrent execution and asynchronous interactions, it was interesting for us to attempt the implementation of these characteristics which require a strict synchronization.

The third and final reason for selecting YAWL was its simplicity with respect to the bindings with external applications. In fact, neither the specification of the language nor the related papers provide concrete details to incorporate external applications into the processes. The only available sources for these details are the source code of the reference implementation, and its manual.

[1]Petri nets are not a workflow specification language and a more appropriate name for the component that we implemented would be Petri nets *interpreter* or *simulator*. However, to maintain a consistent terminology between the case studies, we refer to the Petri nets implementation as the Petri nets *engine*.

To implement an engine for YAWL, the first aspect to be considered are the five dimensions typically involved in a process (see figure 7.2). Although these dimensions are not always mentioned in YAWL documentation, they have been discussed by YAWL's designers [vdAvH02] and they have a prominent place in the design and enactment of YAWL's processes.

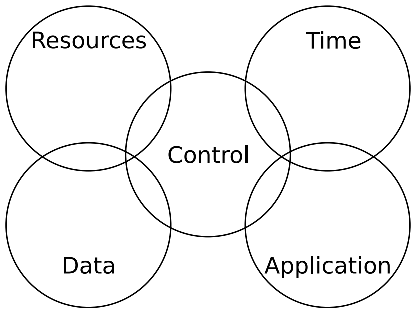


Figure 7.2 The 5 dimensions of a process in YAWL

The five dimensions of a process in YAWL are the following.

1. *Control.* This dimension represents the core of YAWL and defines the tasks in a process and their order of execution. Specifying the elements of this dimension is the central objective of the YAWL language.

2. *Application.* This dimension defines the actual activities to realize in a process. For example, specifying whether an activity must be automatic or must be performed by a human, is something that belongs in this dimension. Therefore, the specification of bindings and bridges to concrete external applications are found in this dimension. YAWL's documentation uses the term *services* for these elements.

   In spite of its importance, in YAWL this dimension is only represented by one element, which is called `Decomposition`: each `Task` in a process should have a `Decomposition` associated which specifies what the `Task` is about.

3. *Data.* This dimension describes the structure of data produced and consumed by a process, and its relation to activities.

   In YAWL's official implementation this dimension has two aspects. On the one side, XSD schemas are used to define the data that a process can manage. On the other side, XPath [CD99] and XQuery [BCF$^+$07] are used to specify the data that each activity is allowed to consume, and to specify how the data produced by each task is transformed and stored.

4. *Time.* This dimension describes timeouts and expiration dates for tasks in a process. In YAWL's official implementation these are also defined using low level XML information.

5. *Resources.* This dimension describes the participants in the processes and the policies to assign them to tasks. This is probably the most advanced dimension in YAWL after control, and its implementation offers powerful characteristics. On the one hand, it allows the external administration of participants, i.e. in an LDAP system. On the other hand it supports a complex life cycle for task assignment to resources: tasks are first offered

to some participants, then they are selected by the participants willing to do them, and finally their execution begins. The mechanisms to define which participants are able to perform the tasks range from very simple (e.g. "Anyone can do the task", or "User X must do the task"), to very complex (e.g. "Users with characteristics Y and Z, and which have not participated yet in this instance of the process"). Furthermore, at any point in time this assignment process can be modified or bypassed by the administrator of the process.

The control dimension is the best documented and best known dimension of YAWL. It also presents the most interesting challenges to Cumbia from the point of view of modeling interaction and state. Therefore, our case study focused on this dimension. Nevertheless, we did not entirely neglect the other dimensions.

Figure 7.3 shows the symbols to graphically represent the elements available in YAWL to model the control flow of processes. The semantics of all these elements are defined in terms of EWF-Nets, and thus they depend on the transfer and consumption of tokens. *Flows* define how tokens can be transferred between elements in a process, and elements are only executed when tokens are available for them to consume (as in Petri nets). There are four main categories of elements in YAWL.

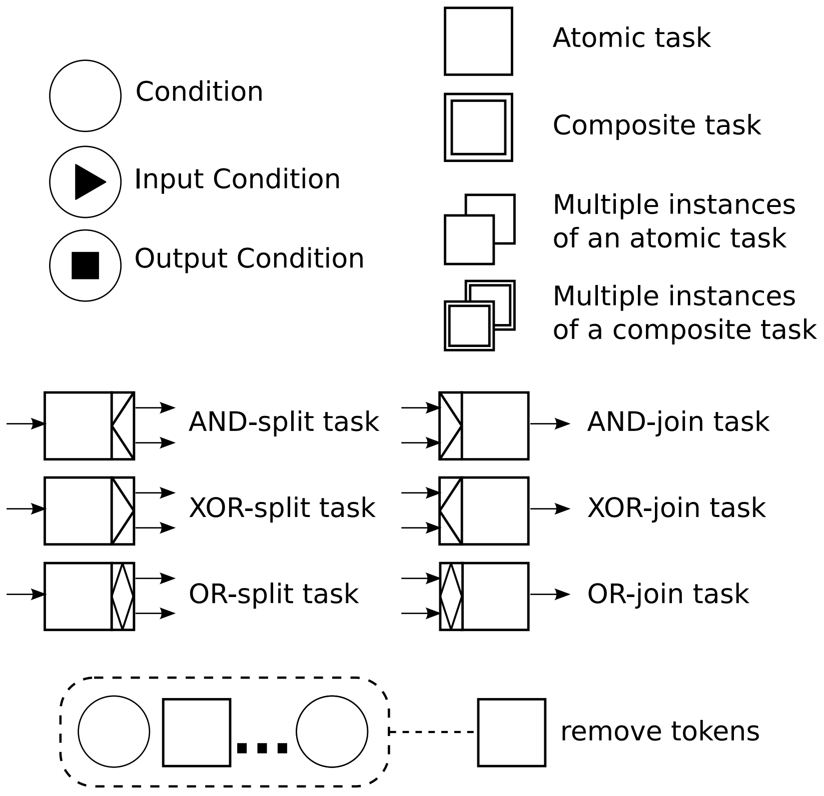


Figure 7.3 Symbols used in YAWL [vdAtH06]

- *Nets.* `Nets` enclose structured sets of elements and have a role similar to `Process` in `MiniBPMN`. `Nets`, however, cannot be nested directly, as in `MiniBPMN` or BPMN.

- *Tasks.* These are the units of work in a `net`. They can be *atomic*, and represent something that has to be performed once. They can also be *multiple*, and represent the same action that has to be performed multiple times, in parallel. Finally, `tasks` can also be *composite* when a `sub-net` has to be performed.

  Furthermore, each `Task` has a *split behavior* and a *join behavior* that determine the interaction of a `Task` with the other elements in the `Net`. For example, a `Task` with an *AND-Join* behavior waits until tokens are available through all the incoming flows to start its execution. When a `Task` with an *AND-Split* behavior is completed, it makes a token available through *all* of its outgoing flows.

- *Conditions.* These are elements of YAWL that can contain one or more tokens, without consuming them. They are analogous to `places` in Petri nets, and they create some of the most complex synchronization problems for an implementation of YAWL. `Conditions` are so named because when they contain a token it means that a certain condition holds in the `net`.

- *Cancellation regions.* A `cancellation region` is a group of `tasks` and `conditions` in a `net` which can be *cancelled* when a certain `task` (located outside the cancellation region) is executed. The cancellation operation happens by removing all the tokens in the region, and aborting the execution of every task that had already started. `Cancellation regions` cannot be easily modeled with Petri nets, and they are one of the main reasons to define YAWL's semantics using EWF-nets.

Implementation details

The implementation of YAWL on top of the Cumbia platform is called YOC (YAWL on Cumbia)[2]. At first, we identified the concerns involved (control, time, resources and data) and we established the metamodel structure shown in figure 7.4. With this division we assimilated the dimensions of control and application because we do not consider it very likely for them to evolve independently. Also, we do not expect to have control models that are reused independently from application models, and vice-versa. Since control is the only dimension in YAWL that has been formally specified, and is the best known one, we decided to only address this dimension initially, by implementing a metamodel for it. Figure 7.5 shows the elements in the control metamodel for YOC. The figure shows that we included in this metamodel the elements to model the application dimension, i.e. we have `decompositions` as an element in this metamodel.

Besides initially leaving out of the implementation the dimensions of time, resources, and data, we also left out one aspect of the control dimension: the *OR-Join*. It has been shown that the semantics of *OR-Joins* in workflow languages are complex, and usually ill-defined, and thus create problems for the implementation of workflow engines [vdADK02]. YAWL is one of the few languages that correctly supports this construct. YAWL's developers have discussed their implementation of this element [WEvdAtH05], but it is quite complex compared to the rest of the

[2]YOC is the main result of the work of Diana Marcela Puentes at the Universidad de los Andes towards her MSc. degree. Diana worked closely with Mario Sánchez in this project and her advisor was Jorge Villalobos.

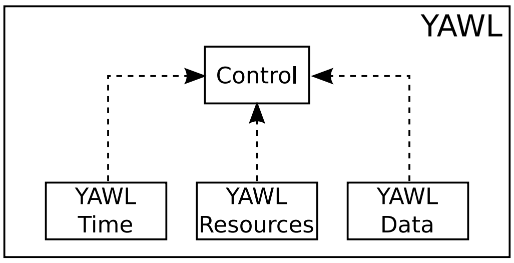


Figure 7.4 Concern specific metamodels in YOC

language. We decided against supporting the *OR-Join* in YOC because it added too much complexity to the implementation without giving us extra information about the modeling capacities of Cumbia and of the open objects. Correctly supporting *OR-Joins* is mostly a matter of algorithmic and its solution is largely independent of the underlying implementation artifacts.

With respect to the syntax of YAWL, we have also introduced two details in our implementation but they do not change at all the semantics of the processes. The first one is to assume AND-Join and AND-Split behaviors for all `tasks` where these behaviors are not explicitly defined. This assumption does not change the semantics of the processes because in YAWL it is mandatory to specify a join behavior for all `tasks` with more than one incoming `flow`, and to specify a split behavior for all `tasks` with more than one outgoing `flow`. Although it is not evident for a process designer, the official YAWL editor assumes the same behaviors that we do, and our implementation only makes this explicit. The second assumption in YOC was to introduce a `condition` in each `flow` that connects two `tasks`. This simplified our implementation as we did not have to consider three kinds of `flows` (task-to-condition, condition-to-task, and task-to-task) but only two (task-to-condition and condition-to-task). This assumption does not change the semantics of the language either. In fact, we are only reversing a decision of YAWL's designers where they decided to hide `conditions` connecting two `tasks` to simplify the layout of YAWL's diagrams. In terms of Petri nets, our decision is equivalent to forcing `flows` to connect `places` and `transitions`, and disallowing `flows` connecting two `transitions`.

We are now going to skip the "easy" details of YOC's implementation, and we are going to discuss the main difficulties that we encountered. The rest of the implementation was straightforward and did not provide information about Cumbia's capabilities beyond what we already knew from other case studies.

Figure 7.6 and figure 7.7 depict two YAWL `nets` that appear simple, but which were not trivial to support in YOC. The crucial characteristic in the first `net` is to have a `condition` followed by *two* `tasks`. The semantics of this structure matches that of the workflow pattern *Deferred Choice* [RtHvdAM06]: after `Task 1` has been completed, both `Task 2` and `Task 3` are *enabled* to be executed. Which `task` is executed depends on an external factor, but in any case *only one* of the two can be executed. The difficulty arises from the facts that both `tasks` can potentially consume the same token, that both `tasks` have to be enabled to be offered to users, and that only one of them can consume the token. Therefore, mechanisms are required to guarantee that both `tasks` are notified about the token's presence, to guarantee that only one of them is activated (no inconsistency because of two activations), and to guarantee that at least one of

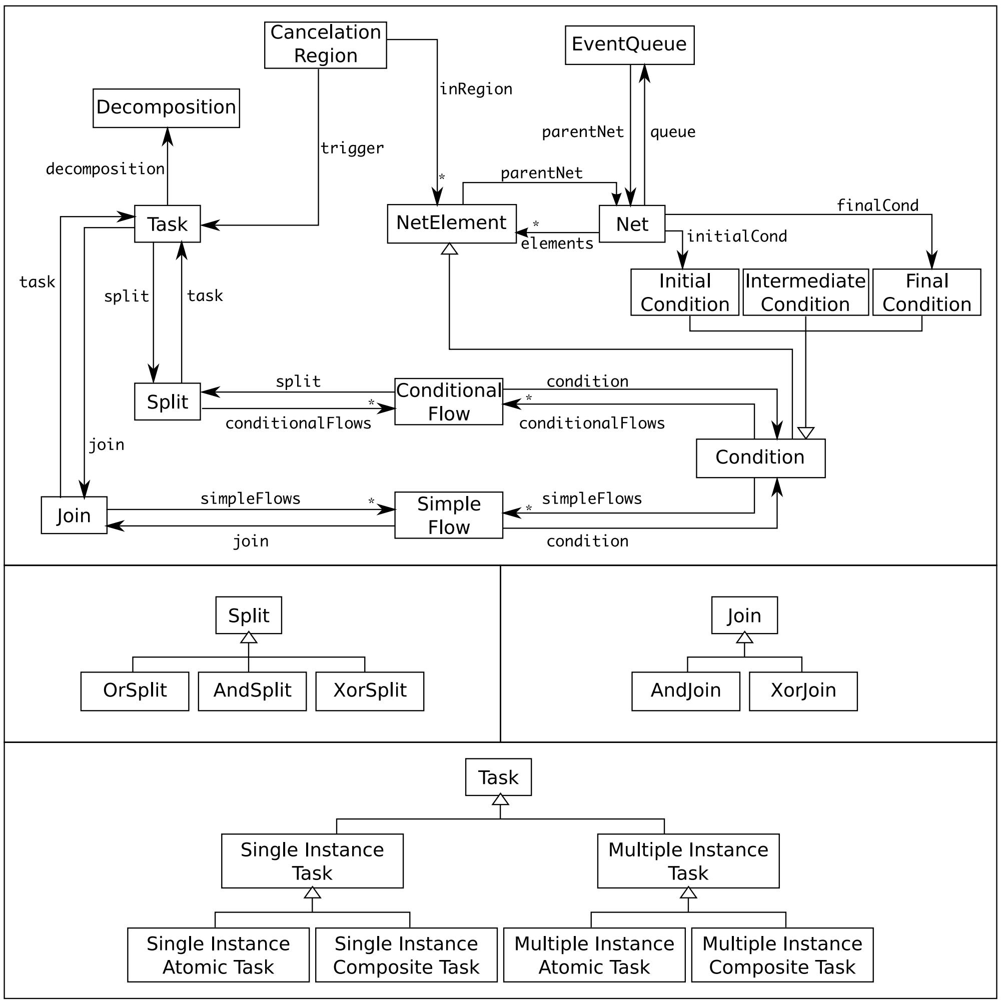


Figure 7.5 The control metamodel for YOC (divided in parts for readability)

them is activated (no deadlocks between them).

The second problematic `net` adds further complications to the previous one. In this case, after `Task 1` is executed, only one among `Task 2`, `Task 3`, and `Task 4` can be executed. Each one of those requires two tokens to work, which can be found in `conditions` `A`, `B`, and `C`. Each `task` shares its incoming conditions with the other `tasks`, and each `task` is enabled at the same time – just after the execution of `Task 1` is completed. With respect to the first `net`, there are two additional problems in this one. Firstly, in the case of attempting to execute the three `tasks`, it is possible to have a triple deadlock if tokens are assigned one by one. Secondly, assigning the tokens to any `task` involves *negotiating* with the other two (and potentially more in more complex nets). This additional complexity brought by the shared `conditions` makes it impossible to apply certain solutions that would work in the first net.

Supporting such `nets` in YOC presented some difficulties due to two main reasons. On the one hand, there is Petri nets' nature as discrete transition systems where only one `transition` is activated at a time, and where the consumption and production of tokens occur instantaneously and at exactly the same time. On the other hand, there are Cumbia's asynchronous coordination mechanisms,

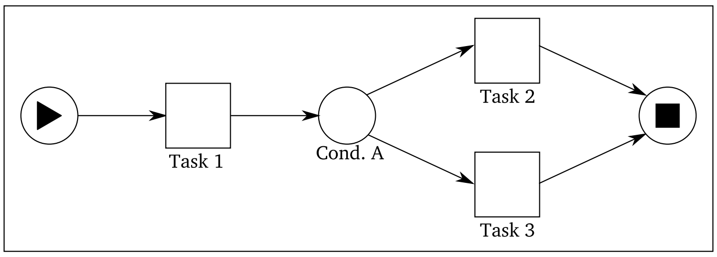


Figure 7.6 Shared condition in a YAWL net

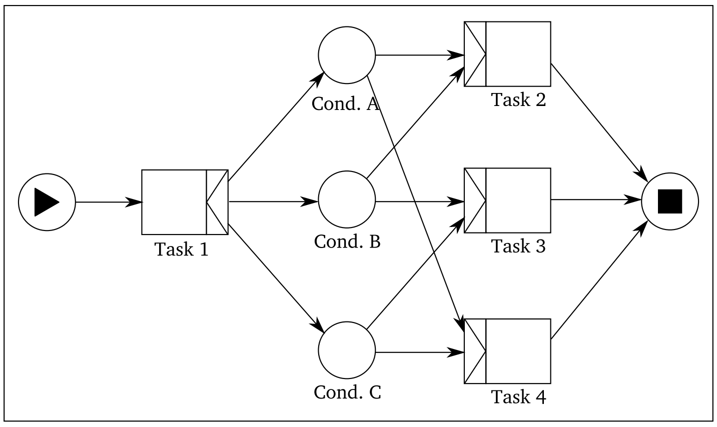


Figure 7.7 Interlocked shared conditions in a YAWL net

where multiple external requests can be received at the same time, and where only one open object can be updated in each moment[3]. Therefore, a Cumbia based Petri net is going to traverse some states that are semantically incorrect. For example, after the activation of a `transition`, there will be moments where only some of the input tokens have been consumed or when some of the output tokens have been produced. Eventually, a consistent state will be reached, but in the meantime other actions can happen (e.g. activating another `transition`, consuming a token) and lead to an inconsistent state. Therefore, an implementation of Petri nets on top of Cumbia should take extra precautions to prevent this kind of inconveniences.

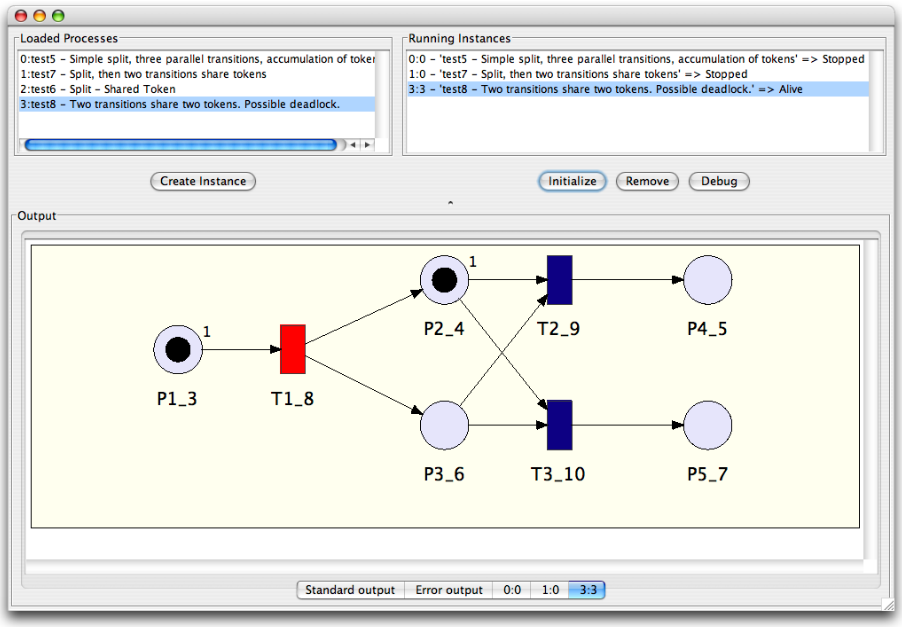
Using Petri nets to represent workflows is known to require an important change to their execution semantics: if `transitions` are used to represent activities, then they cannot be instantaneous and real concurrency and asynchrony has to be introduced. This creates the potential problems previously exposed, namely deadlocks and inconsistencies. This situation has been studied and three solution strategies have been proposed: to introduce a centralized control system, to distribute the control over the places, and to distribute the control among the edges [Bar02]. During the development of YOC we tested two of these strategies.

The first strategy that we tested was to *distribute the control over the places* of a `net`. This strategy basically involves implementing a distributed algorithm

[3]Here the term *moment* is used to refer to the lapse of time required to process one event received by an open object, select the transition to trigger, execute the actions associated to it, emit the necessary events, and update the state of the open object. Therefore, a *moment* can be very brief, when there are no transitions to trigger and the event can be discarded, or can be lengthy if the actions to execute are long lived.

to assign resources to distributed elements. In [Bar02] an algorithm is proposed, but it depends on some characteristics that cannot be guaranteed over open objects. Instead, we adapted the algorithm to create mutual exclusions in networks presented in [RA81], and implemented it on top of open objects.

We did not test this strategy with the full YAWL metamodel. Instead, we built a smaller case study based on Petri nets, and then we extrapolated our conclusions to YAWL. In this case study we first designed and implemented a basic metamodel for Petri nets, and then we modeled the problematic scenarios. As part of this metamodel we implemented the first strategy to correctly execute these scenarios. We also created an interactive client for Petri nets, and a testing environment (see figure 7.8).



0:test5 – Simple split, three parallel transitions, accumulation of toker
1:test7 – Split, then two transitions share tokens
2:test6 – Split – Shared Token
3:test8 – Two transitions share two tokens. Possible deadlock.

0:0 – 'test5 – Simple split, three parallel transitions, accumulation of tokens' => Stopped
1:0 – 'test7 – Split, then two transitions share tokens' => Stopped
3:3 – 'test8 – Two transitions share two tokens. Possible deadlock.' => Alive

Figure 7.8 The Petri nets client showing one of the problematic cases

The Petri nets metamodel was extensively tested using both simple and complex `nets`, uncovering several corner cases where deadlocks or inconsistencies could occur. After we fixed all the problems we ended up with a set of open objects with very complex state machines. The main reason for this complexity was attempting to use asynchronous communications for most interactions, instead of putting a lot of the behavior in synchronized pieces of code within the entities' methods. The main conclusion of this case study was that implementing the strategy using open objects was possible, but not practical: the state machine of the element `Transition` in this metamodel had 5 states and 28 transitions. The same strategy applied to the YAWL metamodel would have required even more complex state machines. Because of this, we rejected the idea of implementing a distributed token assignment algorithm and we opted instead for a centralized solution.

There are at least two ways to implement a centralized control system to solve the execution problems of an interpreter for Petri nets, and both of these ways can be implemented on top of open objects. The proposal of [Bar02] involves defining *locksets* (sets of `transitions` that can compete for the same tokens) and resolving the conflicts at the lockset level. The way in which we implemented a centralized control system is potentially slower because conflicts are resolved at the process level. Nevertheless, the end result is the same: no deadlocks and no inconsistencies are allowed.

We implemented this strategy in YOC and its core is a queue that stores the names of tasks enabled at any given moment. The tasks in this queue are executed one by one and the tokens are assigned atomically. Tasks are also removed from the queue when they are no longer enabled. This guarantees that *i*) no deadlocks are caused by tokens assigned to tasks that never execute, and guarantees that *ii*) no inconsistencies happen because of tasks that execute without having the required tokens. From the point of view of the semantics of Petri nets, this strategy also serves to make a correct implementation: on the one hand, at most one transition can be activated at any given time; on the other hand, the consumption and production of tokens happen in different times, but no other actions can occur before the whole procedure is completed.

The strategy implemented also has two disadvantages to consider. The first one is adding a potential bottleneck for the process execution. This bottleneck is the queue processor, although the time required to process each element in the queue should be very short and thus should have a minimal effect on the duration of a process execution. The second disadvantage is adding in the metamodel an element that is an implementation artifact and does not really have a place on the model. In a subsequent version of this case study we intent to remove the queue from the metamodel and make it part of the engine.

Experiments with YAWL

The first experiment to consider in this case study is the implementation itself of the language. Furthermore, using the CTF we built a test environment for YOC and we used the official YAWL editor to build a test suite. This test suite tested both simple and complex test cases, and the results of these were compared against the semantics displayed by the official YAWL engine.

The second experiment performed with YOC involved its integration with `XTM`. In the beginning of this section we described the time dimension in YAWL, which is very limited. We did not implement this aspect, but instead we successfully integrated the control metamodel to the existing `XTM` metamodel (see figure 7.9). We consider this a successful example of metamodel reuse. Furthermore, we also consider that this experiment shows that a dimension can be enriched by introducing more powerful metamodels: `XTM` makes it possible to describe richer time based restrictions than the basic time dimension of YAWL.

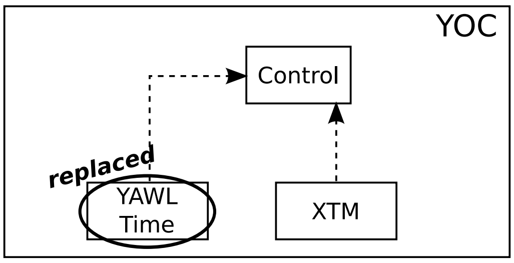


Figure 7.9 Control and time metamodels in YOC

Results and conclusions

In this case study we did an evaluation of Cumbia using two languages with formally defined semantics. The main difficulty encountered is due to the synchronous nature of Petri nets, which demands for the atomic update of several elements' states. This was initially a problem for Cumbia, because most interactions between open objects happen in an asynchronous way through events. Nevertheless, within these scenarios we studied and implemented two alternative strategies to support the required semantics. From these experiments we can conclude that Cumbia will be able to support other workflow languages based on similar synchronous interactions.

The other characteristics and requirements associated to Petri nets and to YAWL did not present relevant problems to be implemented on top of Cumbia. On the contrary, we were able to take advantage of the metamodel flexibility and the language modularization to extend the language in a controlled way. Another advantage was implementing these engines on top of an existing (and tested) platform: thanks to this, the implementation effort was not as much, and it was more focused on the language itself. Unfortunately, we cannot compare the size of our YAWL implementation with the size of the reference implementation because the features supported are not exactly the same (e.g. the reference implementation is tied to a JEE application container).

7.5 PaperXpress

General description

The next case study to discuss is that of PaperXpress[4], an application for a specific domain which combines multiple concern specific languages. Since we established the requirements for the application and we designed the languages, we were able to use this scenario to do interesting validations of Cumbia's expressiveness. In particular, we tested its capacity to support different kinds of language extensions, and, to a lesser extent, its capacity to support dynamic adaptation [SJVD09, Jim07]. In this section we will first discuss the initial implementation of the application as a set of metamodels, and then we will describe the experiments that we did to extend the languages involved.

PaperXpress is an application built using the Cumbia platform to support the definition and execution of processes for writing and reviewing research articles. PaperXpress allows an author to define and perform an ordered set of activities, like writing and reviewing, and relating them with sections in an article. An author starts using PaperXpress, by first defining the structure of the paper he wants to write. This is achieved by using the application shown in figure 7.10. In this image, the author already defined a structure composed by six sections and four subsections.

After the article structure is defined, the author can plan activities to write and review each part. For this purpose, he can use the application shown in figure 7.11, which offers an *ad-hoc* language with very simple control structures. In the sample process shown, the author defined a process with steps to write

[4]PaperXpress is the main result of the work of Camilo Jiménez at the Universidad de los Andes towards his MSc. degrees. Camilo's advisor was Jorge Villalobos.

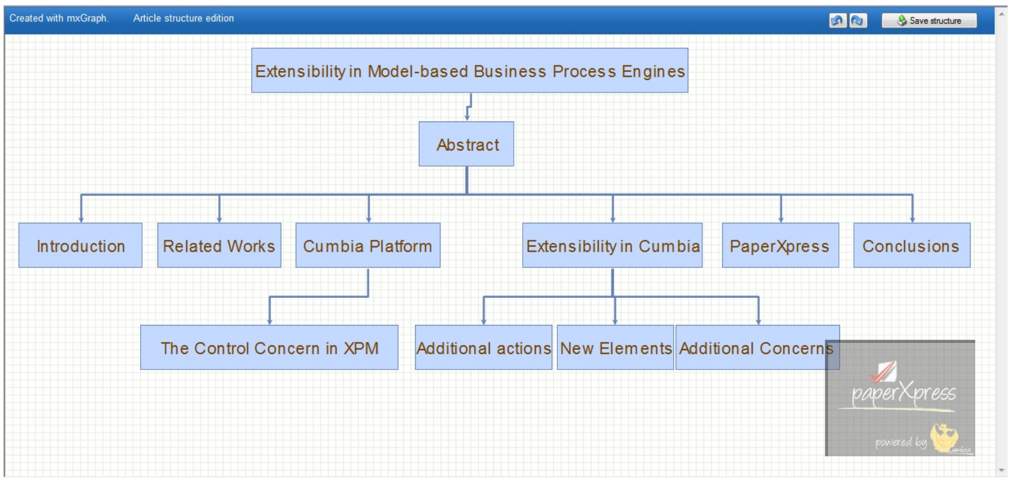


Figure 7.10 The structure of an article in PaperXpress

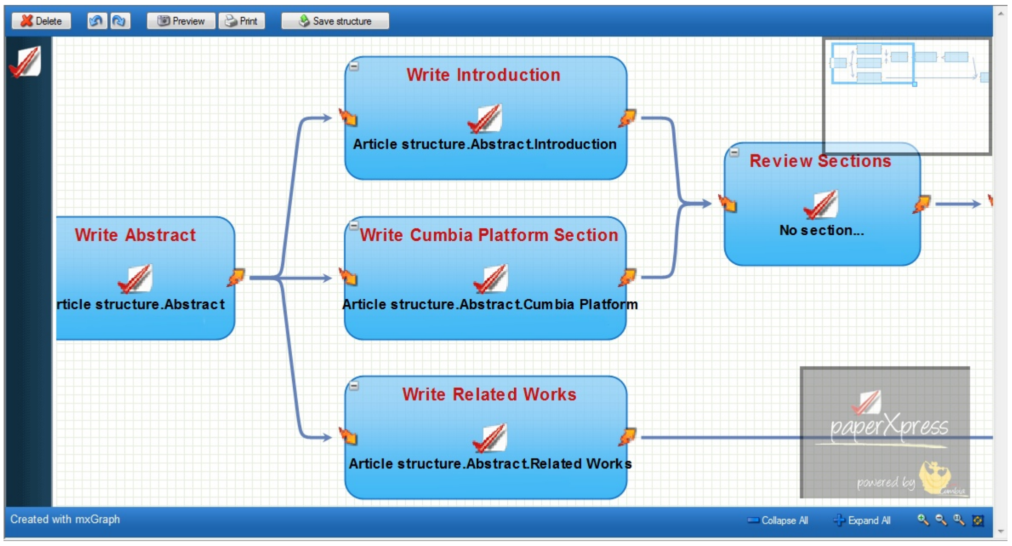

Delete
Preview
Print
Save structure
Write Introduction
Article structure.Abstract.Introduction
Write Abstract
rticle structure.Abstract
Write Cumbia Platform Section
Article structure.Abstract.Cumbia Platform
Review Sections
No section...
Write Related Works
Article structure.Abstract.Related Works
paperXpress
powered by
Created with mxGraph
Collapse All
Expand All

Figure 7.11 A part of the process to write and review an article using PaperXpress

the abstract, write the main sections, review the sections, and finally write the conclusions.

Implementation details

In order to implement PaperXpress using Cumbia we first identified the two principal concerns and we developed the corresponding metamodels, which are shown in figure 7.12. The `PxControl` metamodel, which is used to describe the control concern, has elements to describe processes and activities to write or review parts of an article (see figure 7.13). Since PaperXpress does not require complex control structures, this metamodel is very simple and supports only basic control flow patterns [vdAtHKB03] through three major constructs: `Process`, `PxActivity` and `Flow`. A `Process` always has an initial activity, but the end of the process is implicit (as for the *implicit termination* control flow pattern).

On the other hand, the `PxArticle` metamodel has the elements to describe the structure and the contents of an article (see figure 7.14). These elements

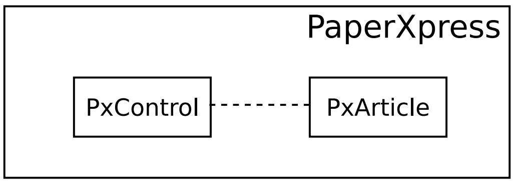


Figure 7.12 Basic metamodels in PaperXPress

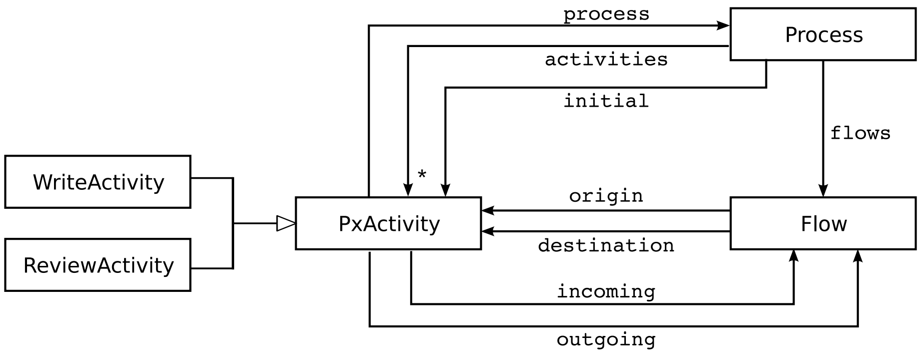


Figure 7.13 `PxControl`: metamodel for the control concern in PaperXpress

include sections and subsections, which give the article a structure, and fine grained content elements, such as paragraphs and images.

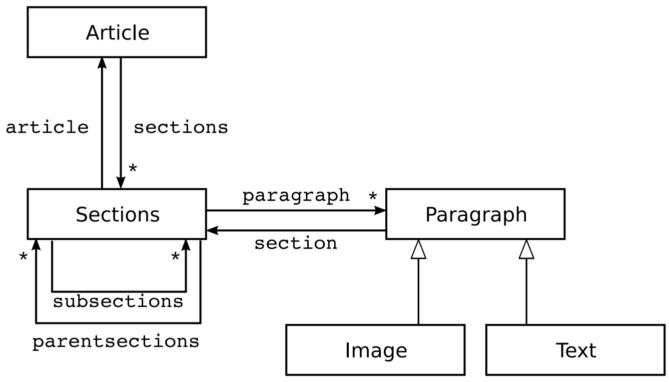


Figure 7.14 `PxArticle`: metamodel for the article concern in PaperXpress

There are several relations between the control and the article concern in PaperXpress, and these can be realized with CCL. One example of those relations is a rule specifying that article sections can only be changed when the related activity is active. In this way, an activity unlocks its section as soon as it starts its execution, and locks it again when it finishes. The coordination between these elements can be achieved with two actions installed in the state machine of the activity, associated to the transitions occurring when the activity starts and ends its execution. These actions notify the corresponding sections to be unlocked or locked.

Another example can be found when the process ends. At that moment, PaperXpress automatically generates a pdf file with the contents of the article using a predefined format. This composition is implemented with actions installed in the `PxProcess` state machine. In this case, an action notifying the `article` element to generate the pdf file is automatically installed in the transition taken when the process ends its execution.

Extending PaperXpress

The development of this case study proceeded in two phases. First, the basic metamodels (`PxControl` and `PxArticle`) were developed. Then, the extensibility capabilities of the open objects were tested by developing some extensions to the base applications. Figure 7.15 shows the relations between the metamodels included in these extensions, which are described in the following paragraphs.

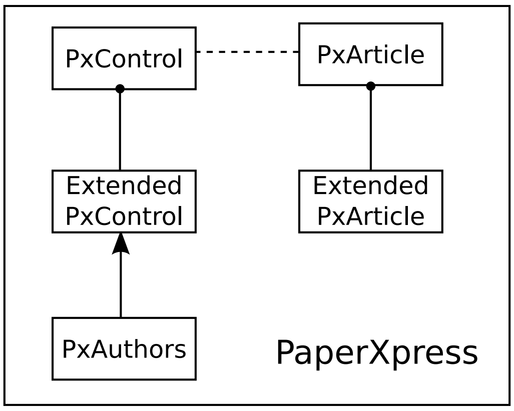


Figure 7.15 The metamodels used in PaperXPress and their extensions

A. Storing the article contents in a remote repository

The first extension to PaperXpress includes a new persistence requirement: the contents of article sections must be kept in a remote repository and they must be checked out only when the activity planned to write or review them is active. Similarly, modifications to these contents must be saved in the repository, when the activity finishes its execution.

To support this requirement it was necessary to extend the behavior of the element `Section` in a new element called `Persistent Section`. More specifically, two actions were implemented and installed in its state machine. A first action, *checkOut*, was installed in the transition taking place when the section is unlocked (see `Persistent Section` state machine in figure 7.16). This action checks out the contents from a specific repository and gives them to the entity. A second action, *save*, was installed in the transition occurring when the `persistent section` is locked. This action retrieves the contents from the entity and saves them in the remote repository.

After installing the new actions, the coordination between the activity and its section changes as follows (see figure 7.16). As soon as a `PxActivity` is activated, its state machine takes the transition from the state *Init* to the state *Active*. This executes the *unlock* action notifying the section that must be unlocked. At that moment, the state machine of the `Persistent Section` takes the transition to the *Unlocked* state, executing the *checkout* action to retrieve the contents from the remote repository. The contents remain available and are changeable until the `PxActivity` is completed. In that moment, the transition to the *Finalizing* state is taken. The action *lock* is then executed, notifying the `Persistent Section`

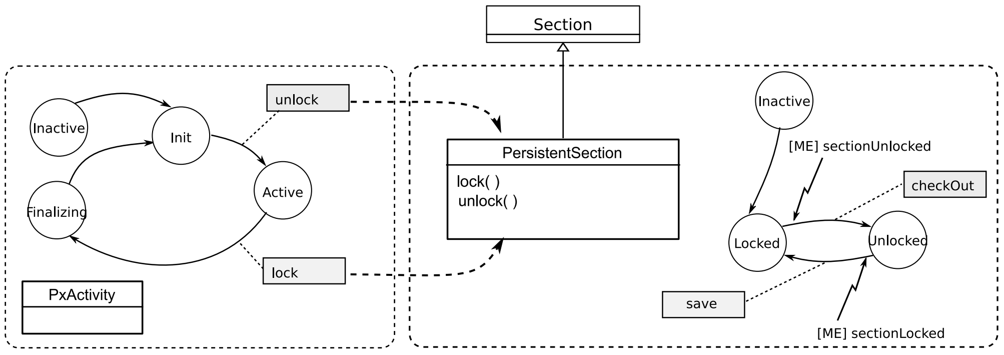


Figure 7.16 New actions in the state machines of a) `PxActivity` and b) `Section`

that it must be locked again. Finally, the state machine of the `Persistent Section` takes the transition to the *Locked* state, executing the *save* action.

Implementing this requirement implied two steps. Firstly, the new actions were implemented in separate Java classes. These classes included the logic necessary to connect and disconnect from a specific remote repository, as well as to save and retrieve contents from it. Secondly, the `Extended PxArticle` metamodel was created, and the element `PersistentSection` was defined using the following snippet of code:

Listing 7.1: Definition of the `PersistentSection` extension

```xml
<extended_type name="PersistentSection" extends="Section">
  <state_machine_extensions>
    <add_actions transitionName="Unlock">
      <action name="checkOut" class="cumbia.actions.CheckOut" />
    </add_actions>
    <add_actions transitionName="Lock">
      <action name="save" class="cumbia.actions.Save"/>
    </add_actions>
 </state_machine_extensions>
</extended_type>
```

B. Revision control for article sections

The second extension to PaperXpress includes a revision control mechanism for article sections. This mechanism must be activated automatically every time an activity modifies a section and it applies the following two criteria: First, a new revision is created if and only if the contents of a section are different from the contents of its current revision. Second, if the differences between those contents are greater than 30%, the new revision should then increase its revision number by 1; this is called a major revision. Otherwise, the revision number is increased by 0.1; this is called a minor revision.

In order to support this new requirement in PaperXpress, we used Cumbia extension mechanisms to allow the inclusion of additional elements to the `Extended PxArticle` metamodel. In particular, two new elements within the article metamodel were defined (see figure 7.17): Firstly, a new element representing the current revision of a section, `Revision`; and secondly, a new extended section composed and coordinated with its revision, `VersionedSection`.

The element that represents the current revision of a section holds the last

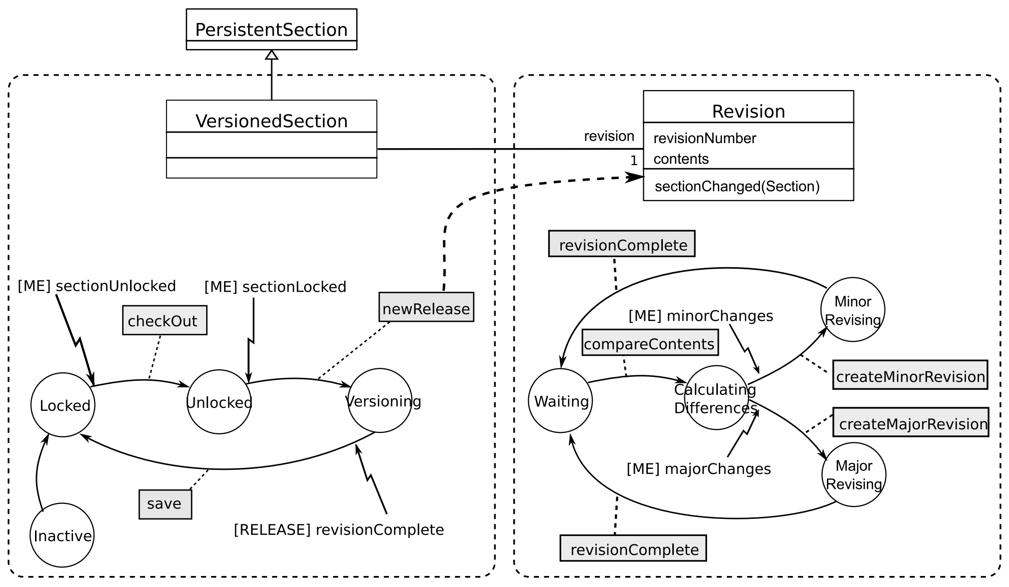


Figure 7.17 New elements in the `Extended PxArticle` metamodel

version of the section contents, and it is identified by a revision number that changes according to the second criterion. Within its state machine, the complete revision process required by the two criteria is represented and coordinated by particular actions that activate the transitions in the state machine. This revision process is triggered by the extended section (`VersionedSection`) when it is locked after an activity has finished. In order to do this, its state machine is being enriched with a new state between the *unlocked* and *locked* states. In this way, when the section is locked it must be revised. As soon as the revision steps finish, the section is locked again.

Implementing this requirement implied three steps. Firstly, the `Extended PxArticle` metamodel was extended to include the new elements and their state machines. Secondly, the new entities (`Revision` and `VersionedSection`) were implemented. They were implemented in separate Java classes. Finally, the new actions were implemented. Changes to the other concerns were not necessary to support these new requirements.

C. Supporting several authors

The third extension to PaperXpress is to support the participation and collaboration of several authors within the process. The authors and their activities must be defined at the same time as the article structure and the process definition. Once they are defined and the process is started, the authors must be able to login into PaperXpress and check their assigned activities. In particular, authors must be assigned one activity at a time. In case an activity is assigned to a busy author (he has unfinished activities assigned), that activity must be queued in his pending activities; pending activities can later be manually assigned.

In order to support the new requirement, we extended PaperXpress with a whole new concern: the authors concern. This concern includes two principal elements in its metamodel: `Author` and `Assignment` (see figure 7.18). In particular, an author might be waiting for an assignment or he might already have begun working on it. These two possible states are represented in the author's state machine, and they define whether or not an activity is marked as pending during process. While an author is waiting, an assignment can be given to him. This assignment is triggered by the predefined process when the corresponding activity starts its execution. At this particular moment, the availability of the author must be checked: If he is busy, the activity is marked as pending and must be activated later manually; otherwise, the assignment is given to the author, and his state changes to *Working*. This assignment policy is represented in the state machine of the `Assignment` element shown in figure 7.18.

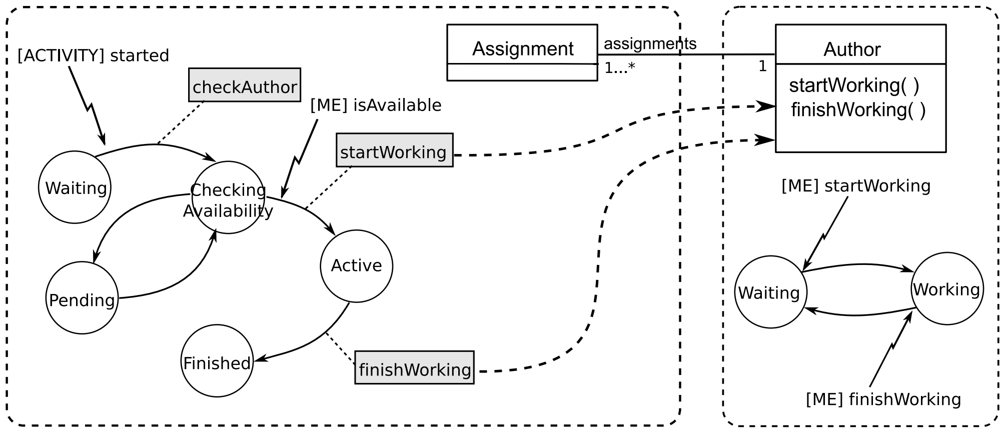


Figure 7.18 Elements in the `PxAuthors` metamodel: a) `Assignment` and b) `Author`

Implementing this requirement in PaperXpress implied three steps. Firstly, the new `PxAuthores` metamodel had to be developed. The Cumbia Editor was used for the metamodel and state machine definitions, and separate Java classes were implemented for the entities and the actions. Secondly, an extension to the `PxControl` metamodel was created. In this extension the state machine of the `PxActivity` has two additional actions that serve to coordinate their execution with the assignments. For each `PxActivity` there is one `Assignment` and corresponding `Author`, and as soon as the state machine of the `PxActivity` changes its state to *Active*, an action notifying the corresponding assignment is executed. The assignment policy is then triggered and the author can start working, or the assignment can be marked as pending. The other action installed in the state machine of the `PxActivity` notifies the assignment to be finished when the activity is completed. The client applications of PaperXpress had to be extended as well to support this extension. In particular, the process definition editor required the most changes.

Results and conclusions

The focus of the PaperXpress scenario was to study the expressive power of Cumbia and its extensibility capabilities. This scenario was scoped to a very specific domain and we had the liberty of defining the languages ourselves. Therefore, we defined these languages in ways that were adequate for Cumbia and we did not encounter major problems in their implementation. We did not enjoy the same advantage in the IMS-LD scenario that is presented in the next section. In spite of this, PaperXpress illustrates the core characteristics of the platform.

Among the metamodels used in PaperXpress, only PXControl has some similarities to the languages used in the other case studies. The other metamodels are very different as they mostly focus on state and have a relatively simple behavior. If Cumbia were based on a unique workflow execution model or an intermediate language, supporting those other languages would probably be very difficult. On the other hand, the experiments performed in this case study also provide evidence about the advantages brought by characteristics **C2** and **C3**. These experiments involved different types of changes and extensions to the languages which were implemented with changes to the metamodels and with the introduction of new ones. In both cases, they illustrate the tight relation between languages and metamodels.

7.6 IMS-LD

General description

IMS-LD is a specification for defining workflows in an e-learning context, using components and learning materials conforming to several complementing specifications [IMS03b] (see section 2.2.1). In this context, IMS-LD is the central specification that is used to define *learnflows*, while the other complementing specifications serve to describe the interaction with other applications and the structure and contents of learning materials. This case study is also based on a domain specific language, but unlike the PaperXpress scenario, in this one the language already existed and was out of our control. Therefore, its semantics had to be implemented as per the specification and we could not adapt its requirements to make them more adequate for Cumbia. Also, we had to support requirements that are specific to the e-learning context and that we had not considered in the previous case studies.

For example, in this domain it is fundamental to offer support for dynamic adaptation because instructors should be able to modify the learnflows, at run time, in response to the students' performance. Another difficulty is the existence not only of activities that every student has to perform independently, but also of activities that require the joint work of several students. Furthermore, the close relation between IMS-LD and other specifications to integrate external services, applications, and learning materials, created some additional restrictions for the implementation. An evidence of the difficulties that these requirements pose is the relative lack of tools to support the IMS-LD specification: at the time when this case study was developed, there was only one full implementation of the specification.

As a result of these characteristics, IMS-LD turned out to be a very useful scenario to test the Cumbia platform. Because of its special characteristics, the platform had to be improved in many ways to accommodate features discovered during the development of the project. In this section we will describe how this case study was developed, the aspects of the scenario that created the most interesting challenges, and the solutions proposed for those challenges.

Implementation details

Garabato [CV08] is the name of the IMS-LD implementation on top of the Cumbia platform[5]. Following the guidelines proposed by Cumbia, Garabato is

[5]Garabato is the main result of the work of Nadya Calderón and Carlos Vega at the Universidad de los Andes towards their MSc. degrees. Their advisor was Jorge Villalobos.

based on several concern specific metamodels, and includes an engine for each one of those. These metamodels are depicted in figure 7.19 and they are described in the rest of the section[6]. The full Garabato application includes an engine for each of the metamodels involved, a client to allow the participation of users (both students and support staff), an application to monitor the execution of the learnflows, and an application to convert IMS-LD specifications into Garabato-compatible model specifications.

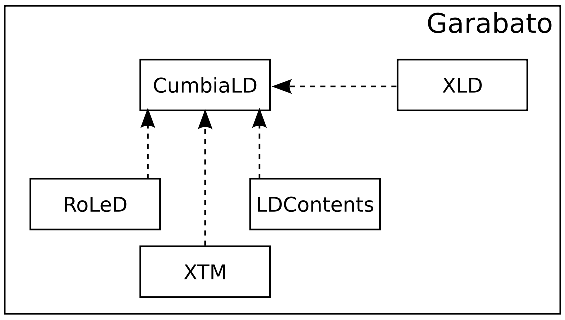


Figure 7.19 Metamodels in Garabato

The central metamodel in Garabato is called `CumbiaLD`, and it embodies the elements required to describe the structure of the learnflows. The basic elements of this metamodel are shown in figure 7.20 and they follow the same structure that was presented in the conceptual model of IMS-LD (see figure 2.4). `CumbiaLD` only targets the level A of the IMS-LD specification.

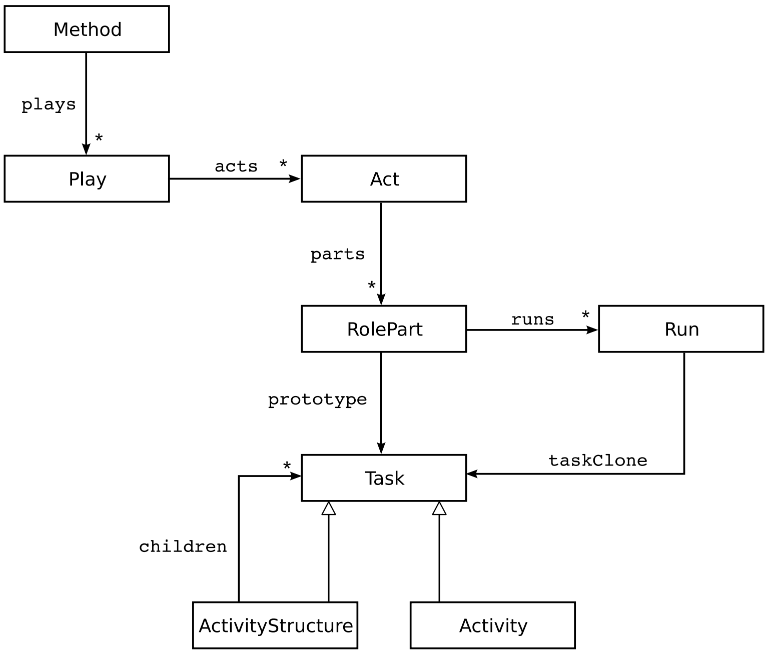


Figure 7.20 Elements in the `CumbiaLD` metamodel

[6]Since the time when Garabato was developed, the Cumbia platform has evolved in several ways, thus making the original implementation of Garabato obsolete. Because of this, Garabato includes some elements that are not allowed in Cumbia anymore, and its metamodels take less advantage of open objects's extensibility mechanisms.

The main issue encountered in the implementation of this metamodel was related to the number of instances of each task that had to be executed for each instance of the learnflow. The exact number of task instances is usually known at run time, and a mechanism to dynamically create new task instances had to be included. This was implemented using prototypes that were manually *cloned* when the number of required instances was known. We later refined this mechanism and adopted it in the implementation of `XPM`'s `MultiActivities` (see section 7.7.1).

Cloning prototype tasks was an effective means to achieve the required result of having several instances of the tasks, but it required some additional things to work properly. First of all, when this functionality was implemented, the developer of the `CumbiaLD` metamodel had to modify the structure of the model and connect the new elements adequately. This included creating the new instances of the tasks, and connecting them to the other elements in the model. However, the single most difficult point was creating the event subscriptions that the new elements required to interact with the other elements in the model. To tackle theses difficulties, newer versions of the Cumbia platform offer the means to *clone* model elements much more easily.

Another related issue had to do with the interaction of the cloned elements with the other metamodels. Since the relations between elements in model instances are created when the models are instantiated and the CCL programs are executed, recently cloned elements were not connected to other models. Therefore, the relations between models had to be manually updated whenever a new element was cloned. Up to the current version of Cumbia and of CCL we have not really tackled this problem. However, we have in our future work plans to introduce finer grained events into CCL, to allow the execution of CCL blocks whenever new elements are dynamically added or removed from the models (currently CCL only supports coarse grained events that refer to the creation or the deletion of model instances).

The `CumbiaLD` metamodel only supports the specification of the control concern for the level A of the IMS-LD specification, but it is complemented by the optional metamodel called `XLD`. The `XLD` metamodel was developed in two stages: at first, it only included the elements necessary to support IMS-LD level B; afterwards, it was enriched with the `Notification` element to support IMS-LD level C. Figure 7.21 depicts the elements in this metamodel. Some of these are shaded in gray to represent that they are not open objects (they are *non-open objects*), although they are part of the models. The decision to handle these elements as simple objects, and not as open objects, was due to their nature as containers without relevant behavior or interactions with other elements.

The other metamodels involved in Garabato did not present important challenges in their implementation. `RoleD` is the metamodel where IMS-LD users and roles are managed. Since the IMS-LD specification defines *learner* and *staff* as the mandatory roots of the hierarchies of roles, this information was included in the metamodel structure (see figure 7.22). The element `User` in this metamodel is also a non-open object, as it only holds *static* information about the users.

IMS-LD also defines the structure of several kinds of resources associated to the learnflows. These resources are generically called *learning materials* and, besides their contents, they also include information about their role in the context of a learnflow. The `LDContents` metamodel was defined to describe learning

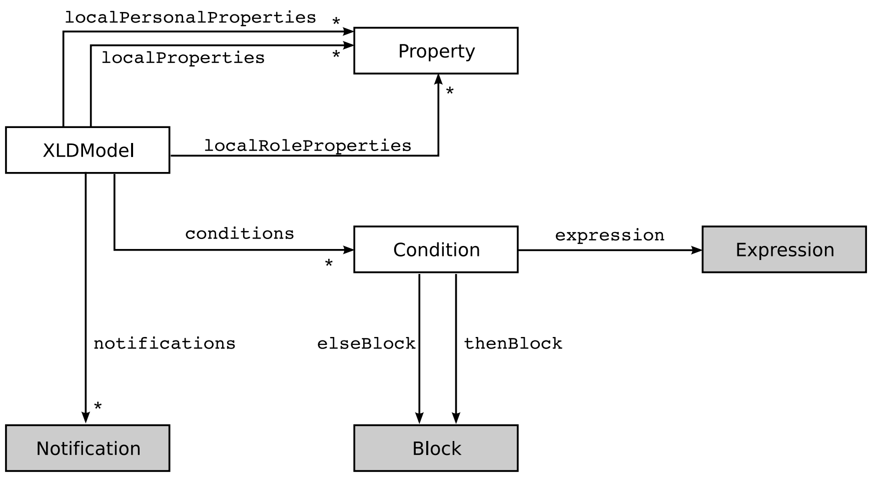


Figure 7.21 The XLD metamodel to support levels B and C of IMS-LD

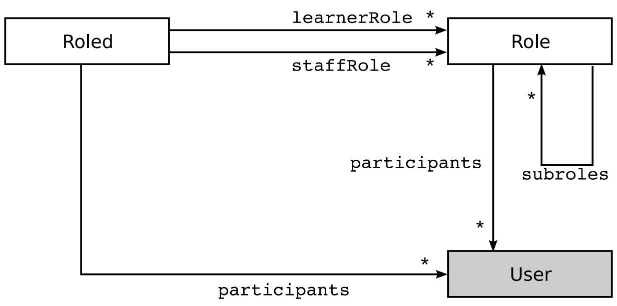


Figure 7.22 The `RoleD` metamodel

materials, and it supports all the elements required by the IMS-LD and IMS-CP [IMS03a] specifications. Figure 7.23 shows the elements of this metamodel. It should be noted that, unlike other metamodels that mixed open objects and non-open objects, in this metamodel all the elements are non-open objects.

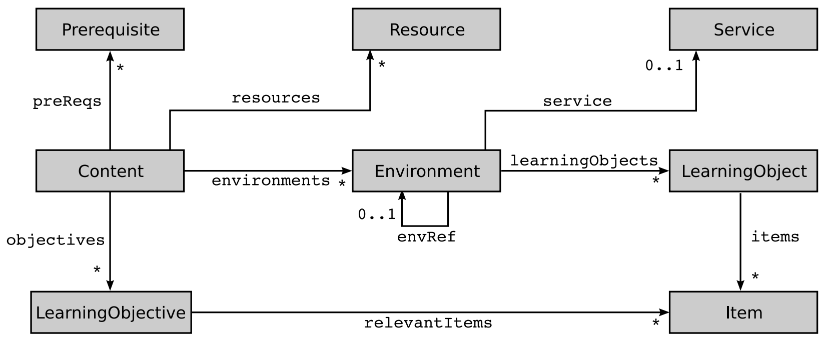
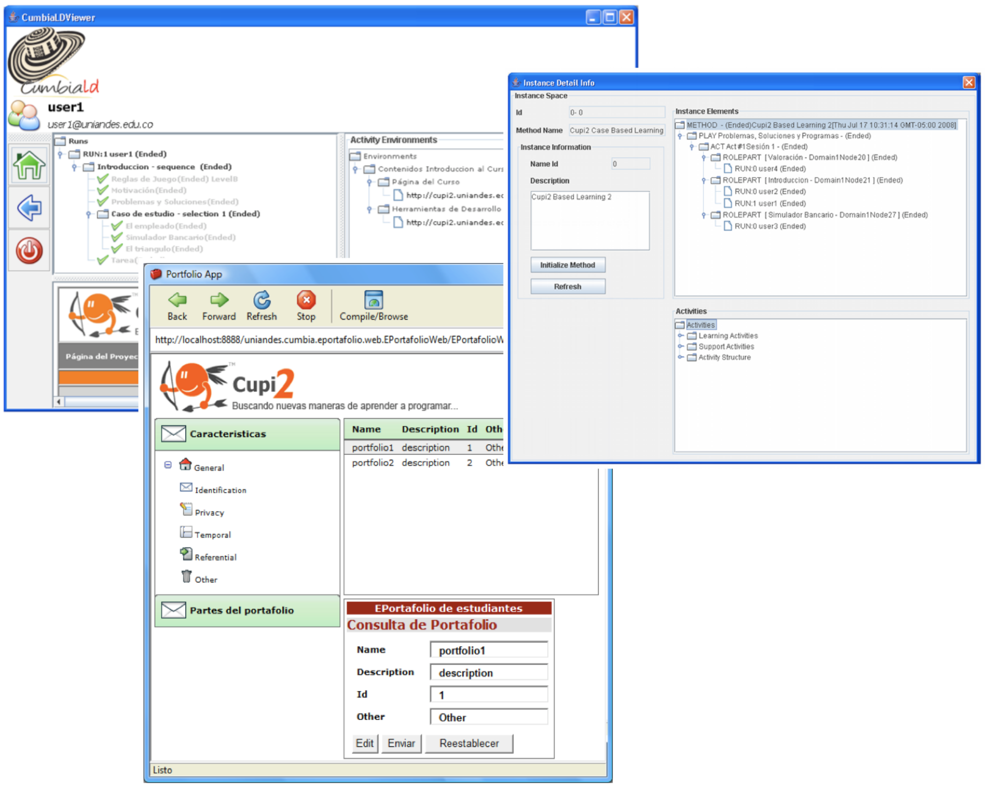


Figure 7.23 The `LDContents` metamodel

Finally, Garabato also employed `XTM` to describe time restrictions over the learnflows.

Experiences with IMS-LD

The experiments related to this case study are basically two. Firstly, there is the implementation of Garabato, which included the development of the engines, of a client for participants in the learnflows, of a console for monitoring the learnflows at run time, and an application to convert IMS-LD specifications into model specifications. Figure 7.24 shows screenshots of the client application and of the monitoring console. This implementation was accompanied by the development of a test suite built on top of the CTF.



```
RUN:1 user1 (Ended)
  Introduccion - sequence (Ended)
    Reglas de Juego(Ended) LevelB
    Motivación(Ended)
    Problemas y Soluciones(Ended)
    Caso de estudio - selection 1 (Ended)
      El empleado(Ended)
      Simulador Bancario(Ended)
      El triangulo(Ended)
    Tarea(
```

```
Environments
  Contenidos Introduccion al Curs
    Página del Curso
      http://cupi2.uniandes.ec
    Herramientas de Desarrollo
      http://cupi2.uniandes.ec
```

http://localhost:8888/uniandes.cumbia.eportafolio.web.EPortafolioWeb/EPortafolioW

Name	Description	Id	Oth
portfolio1	description	1	Othe
portfolio2	description	2	Othe

EPortafolio de estudiantes
Consulta de Portafolio

Name	portfolio1
Description	description
Id	1
Other	Other

Id	0- 0
Method Name	Cupi2 Case Based Learning
Name Id	0
Description	Cupi2 Based Learning 2

```
METHOD - (Ended)Cupi2 Based Learning 2[Thu Jul 17 10:31:14 GMT-05:00 2008]
  PLAY Problemas, Soluciones y Programas - (Ended)
    ACT Act #1Sesión 1 - (Ended)
      ROLEPART [Valoración - Domain1Node20] (Ended)
        RUN:0 user4 (Ended)
      ROLEPART [Introduccion - Domain1Node21] (Ended)
        RUN:0 user2 (Ended)
        RUN:1 user1 (Ended)
      ROLEPART [Simulador Bancario - Domain1Node27] (Ended)
        RUN:0 user3 (Ended)
```

```
Activities
  Learning Activities
  Support Activities
  Activity Structure
```

Figure 7.24 Some screenshots from Garabato

The other part of the experiments performed in the IMS-LD scenario had as goal to test the extensibility of Cumbia. In these experiments Garabato was complemented with the necessary components to support student portfolios[7]. In the e-learning context, a portfolio is a structured collection of information related to a student, which can include information about his grades and the results of his projects and homeworks. There are several complementing specifications close to IMS-LD that define frameworks to structure and manage portfolios: what was implemented in this experiment was a combination of several specifications that include IMS Learner Information Packaging (IMS LIP) [IMS01], IMS ePortfolio [IMS05], and IMS Reusable Definition of Competency or Educational Objective (IMS RDCEO) [IMS02].

Similarly to the elements in the `LDContents` metamodel, the elements in a

[7]The implementation of portfolios into Garabato was the main result of the work of Gilberto Pedraza at the Universidad de los Andes towards his MSc. degree [Ped09]. Gilberto worked in this project with Nadya Calderón and Mario Sánchez, and his advisor was Jorge Villalobos.

portfolio have a meaningful state but do not have a very interesting behavior. Furthermore, each portfolio is associated to a single user, and must be updated only by activities that involve the respective user. Since this was much more complicated than other scenarios where we had dealt with the data concern, we proposed a very different approach: instead of creating a portfolio metamodel, creating a variable portfolio model for each user[8], and then creating a single instance of each model, we opted for the creation of the DMM metamodel. This new metamodel, which is shown in figure 7.25, is a generic metamodel to describe data models. In this case, by generic we mean that DMM can be used in many domains and is not restricted to the e-learning data. Conversely, models conformant to DMM are indeed domain specific, and they structure the information that is relevant for a domain. In the case of Garabato, the DMM model created included all the elements required in a portfolio. The behavior of these models, which is defined in the DMM metamodel, is mainly related to the storage, query, and update of data. In figure 7.25 many of the elements appear shaded in gray (non-open objects) because they are either stateless with complex behavior, or just stateful without relevant behavior.

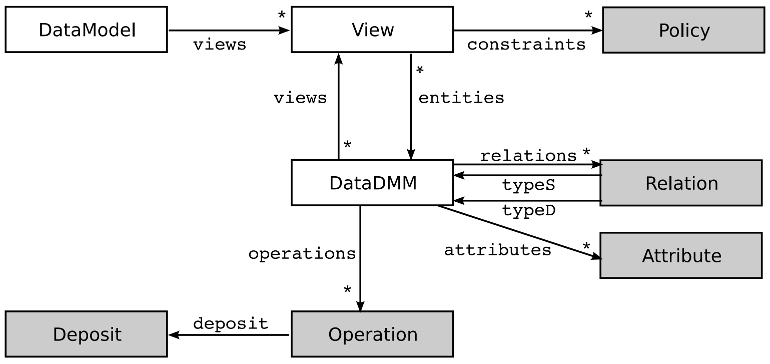


Figure 7.25 The main elements of the DMM metmaodel

Results and conclusions

The implementation of Garabato was a very useful case study for Cumbia that tested aspects such as its expressive power, the flexibility of the languages, and the capabilities of the platform interfaces. Implementing a complex existing language, with a very different structure from most workflow languages, and with the specific requirements of the e-learning domain resulted in new requirements for the platform. Most of these have since been implemented, assimilated, and used in the engines of other languages.

Among the requirements not yet implemented, we believe that the most pressing are those related to the limitations of CCL to handle dynamic adaptation of the models. Therefore, we have identified as work for the near future the design and implementation of the extensions to CCL that will allow the weaver to react to fine grained events produced when the structure of models change.

Finally, a very important result of this case study is to have a complete implementation of the language. Garabato implements all the core features required by

[8]The structure of a portfolio depends on the courses that a student has followed. Therefore, it is not possible to define a single portfolio model and then reuse it for many students.

the language specification, including the dynamicity defined in level C. As it was previously mentioned, this is not supported by most existing implementations of IMS-LD.

7.7 Other experiments

7.7.1 XPM

XPM [SVR09], which stands for eXtensible Process Metamodel, was the first workflow language implemented in Cumbia, and its development drove the development of many characteristics of the platform[9]. XPM focuses exclusively on the description of the control flow, but it can be composed with other metamodels to complement a process description. Furthermore, when XPM was designed an effort was made to keep the number of elements in it as low as possible. The result was a fairly small language with only 8 basic elements (see figure 7.26). Notwithstanding, XPM is very expressive and it supports most of the original control flow patterns [vdAtHKB03], albeit it is somewhat verbose to express certain workflow definitions. The only pattern not supported is *implicit termination*, because all XPM processes explicitly state their termination conditions. Incidentally, this pattern is neither supported in YAWL, whose designers were the proponents of the patterns.

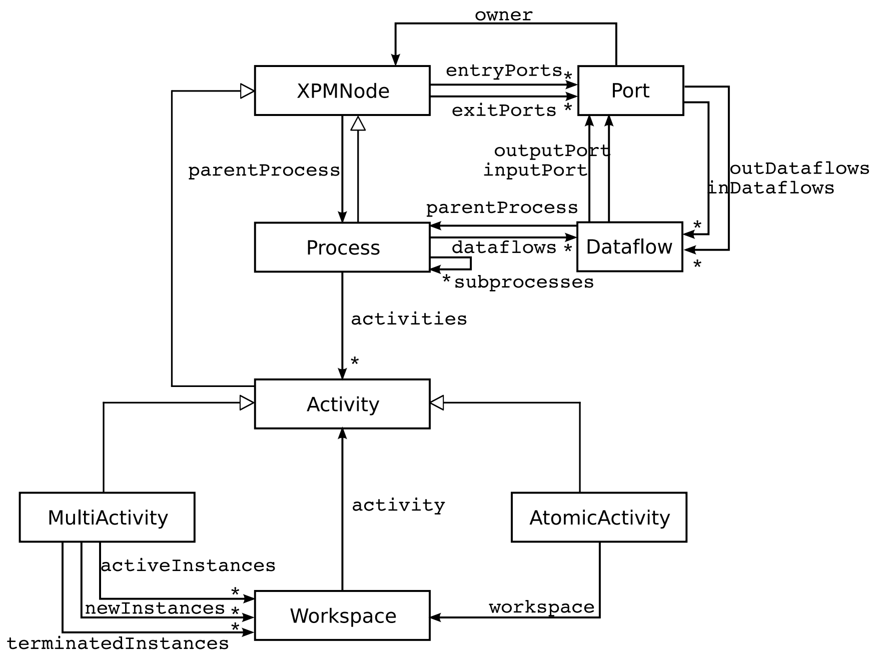


Figure 7.26 Metamodel of XPM

Figure 7.27 shows a simple process modeled using XPM, and it can be seen that XPM's design takes many ideas from the Abstract Process Machine of APEL [DEA98, EVLA+03]. The top-level element of any XPM model is always an instance of Process (Credit Application). To be executed, a process needs some input data which is received through one of its entry Ports. Conversely, when a

[9]The design and development of XPM has benefited from the work of several former participants of the Cumbia project, including Pablo Barvo, Gabriel Pedraza, and Jaime Solano.

process finishes its execution, output data is produced and external entities can find the data in one of the process' exit ports. Within a process, one can find activities which perform concrete tasks (Consult Credit Rating, Evaluate Request, etc.), and other processes (XPMNodes). For each one of these elements, the scheme <*entry ports - element - exit ports*> is repeated as for top level processes. The structure of a process is completed by Dataflows, which connect exit ports to entry ports and thus allow for the exchange of output and input data. The structure of dataflows, entry ports and exit ports in XPM, is a simplified version of the structure proposed in APEL.

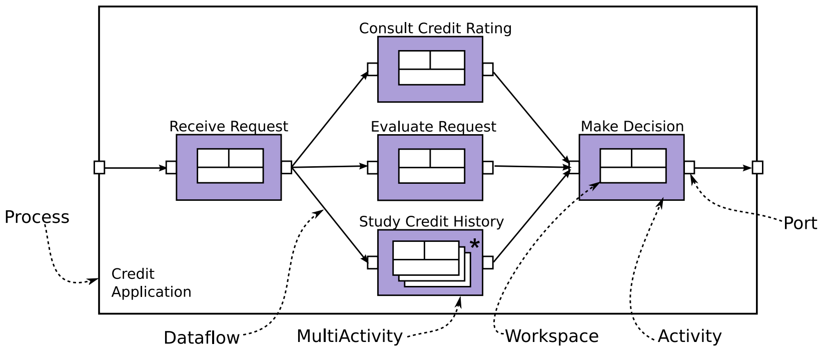


Figure 7.27 A process modeled with XPM

There are three main kinds of elements that can be found as nodes in a process. The first one are processes themselves, and this means that a hierarchy of processes can be created with XPM. The second one are AtomicActivities, which execute specific tasks, and which manage the input data that it requires to run, and the output data that it produces. The specific tasks performed in an AtomicActivity are encapsulated in elements of type Workspace: Workspaces have a very basic behavior and therefore they have to be specialized for specific applications. The third kind of elements in a process are MultiActivities, which hold many similarities with AtomicActivities. The critical difference is in the number of workspaces executed: while an AtomicActivity executes a single instance of its Workspace, a MultiActivity can concurrently execute several instances of the same Workspace. In this respect, MultiActivities are similar to YAWL's Multiple Tasks, or BPMN's Parallel Tasks, although the rules to complete the execution of a MultiActivity are slightly different in XPM.

There are two further characteristics of XPM. The first one is that each XPMNode can have several entry ports and several exit ports. The second one is that each port can have several dataflows connected to it. These two characteristics are sufficient to implement complex control flow routes. Figure 7.28 shows how to represent some common control flow routing patterns using XPM. As a result, additional elements similar to BPMN's Gateways are unnecessary.

From the point of view of the workflow language designer, XPM is not a very advanced language. Nevertheless, it presents the core characteristics shared by most workflow languages, and presents a very interesting scenario to test Cumbia's characteristics. Although XPM has not many elements, it is capable of modeling control flow routing (including advanced patterns), modeling concurrency, and modeling synchronization. Furthermore, XPM models can perform specific tasks which are clearly encapsulated in workspaces' extensions. In the case of MultiActivities, XPM also stresses the capabilities of the Cumbia platform.

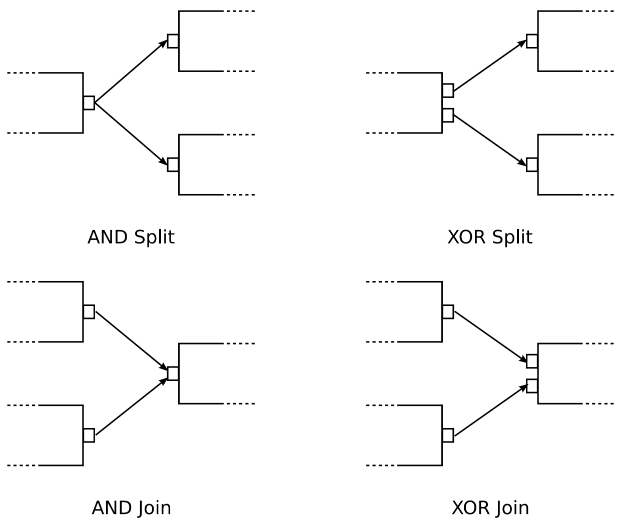


Figure 7.28 Control flow routing in XPM

Since the exact number of workspaces to run in a `MultiActivity` may not be known at design time, XPM's models requires some dynamic adaptation capabilities in order to create new `workspace` instances at run time via cloning (see section 7.6). In the next section we will see how the genericity of this language has been beneficial to use XPM as an intermediate workflow language.

XPM includes several features that are quite different from other workflow languages and that are possibly difficult to implement in other workflow platforms. In particular, the reliance on data flows, and the management of multiple concurrent instances of activities is something that is not normally found in workflow engines. The characteristic **C1** of the platform contributed to avoiding this problem in this scenario. With respect to characteristics **C2** and **C3**, we can comment that the metamodel implementation reflects the structure of the language, and that we have also done experiments where we extend this language and its metamodels. As in other scenarios, we composed XTM models of time restrictions with XPM processes.

7.7.2 BPMN

In chapters 1, 2, and 3 we discussed about the limitations and problematic aspects of workflow engines based on intermediate languages. We encountered some of those probems when we did a case study on the implementation of an intermediate language based on Cumbia. In this case study, called Alegre [Már08], we implemented an engine for a subset of BPMN using XPM as an intermediate language[10]. In addition to providing information about using intermediate workflow languages, Alegre also evidences that Cumbia can support the control flow concern of languages similar to BPMN.

Figure 7.29 shows three aspects of the approach taken to model BPMN on top of XPM. On the one hand, we built a metamodel for BPMN (MM BPMN) where we included all the elements of the language. We also modeled the state machines

[10]Alegre is part of the results of the work of Pablo Márquez at the Universidad de los Andes towards his MSc. degree. Pablo's advisor was Jorge Villalobos. Diana Puentes also participated in Alegre.

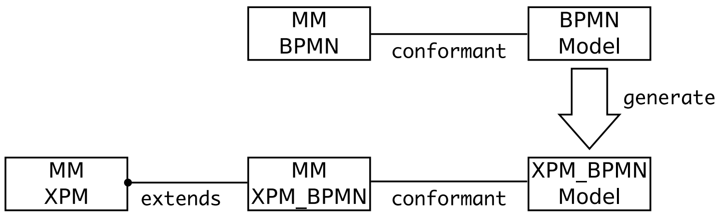


Figure 7.29 Metamodels in Alegre

for these elements, but we only included the states that are relevant from the perspective of a BPMN client. For example, tasks only included the states *Active* and *Inactive*. Furthermore, the most important BPMN interfaces were modeled (e.g. tasks offered services to receive data and to provide data), although their behavior has not directly implemented. The final result of this was a metamodel that was usable to create models, to query the state of models, and to receive BPMN requests, but was not executable on its own.

The second aspect of Alegre was to extend the `XPM` metamodel to implement the semantics of BPMN elements (`MM XPM_BPMN`). For example, a specialized `workspace` that acted as an `XOR-Gateway` of BPMN was created. This extended `workspace` is capable of evaluating conditions associated to flows, and selecting only one of them to proceed with the process. The extended metamodel served to establish a mapping between BPMN models and `XPM_BPMN` models. Figure 7.30 shows a sample from this mapping that involves the aforementioned specialized workspace.

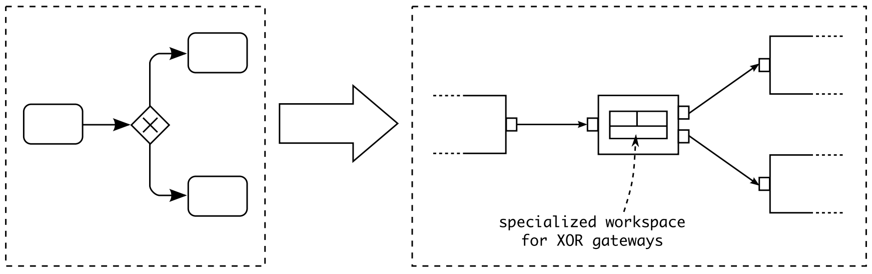


Figure 7.30 Mapping of a BPMN XOR-Split into `XPM_BPMN`

The final aspect of the approach was to build an environment to execute the BPMN models. This first required the construction of a program that used the aforementioned mapping to generate `XPM_BPMN` models from BPMN models. On the other hand, this also involved the creation of an architecture to execute, in a synchronized way, BPMN models and their corresponding XPM_BPMN models. This architecture is called Alegre and is schematized in figure 7.31.

The top part of figure 7.31 shows a BPMN engine built around the Cumbia Kernel and configured with the BPMN metamodel. This engine is capable of receiving requests and interactions from clients that interact with the BPMN process, or users participating on it. This engine can also be queried about the state of the process: since every element has a state machine, answers to these queries can be found by inspecting the state of the open objects.

The bottom part of the figure shows the `XPM` engine used as an engine for `XPM_BPMN`. This engine has access to external systems that support the processes (i.e. data sources, legacy applications, etc.) but it does not receive direct requests

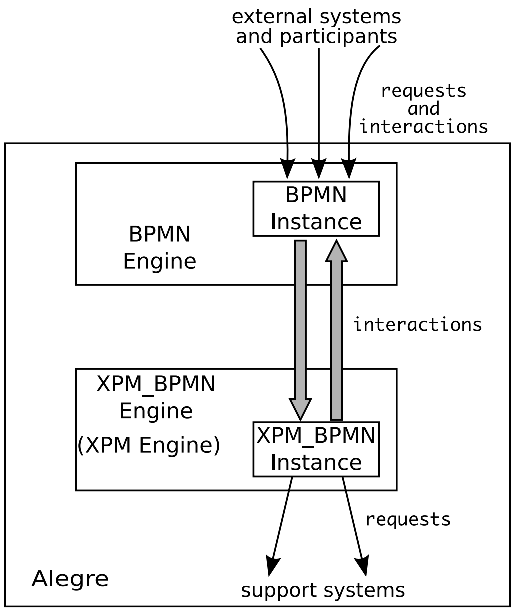

external systems and participants
requests and interactions
BPMN Engine
BPMN Instance
interactions
XPM_BPMN Engine (XPM Engine)
XPM_BPMN Instance
requests
Alegre
support systems

Figure 7.31 Alegre structure and run time behavior

from clients or participants. Instead, this engine is controlled by requests and events generated in the BPMN engine in response to external interactions. In response to the requests and events, actions are executed in the `XPM_BPMN` model, the state of the model is updated, and notifications are sent back to the BPMN model. Finally, these notifications serve to update the state of BPMN elements accordingly to the semantics of the language.

As with most intermediate languages, the presence of `XPM_BPMN` is transparent to participants in the BPMN process. However, from the point of view of the implementor of the engine, this architecture created a few difficulties. On the one hand, the semantics of BPMN still had to be implemented, although it ended up scattered across the BPMN metamodel, the `XPM_BPMN` metamodel, and even the model generator. Compared to the scenarios where the semantics of the languages are packaged inside metamodels, we consider that the Alegre solution is more difficult to maintain and to evolve. Another problem is that three different mappings had to be considered: the first one served to convert BPMN models into `XPM_BPMN` models; the second one served to convert requests from BPMN to `XPM_BPMN`, and the third one served to convert `XPM_BPMN` state updates and responses into BPMN state updates and responses. Creating those mappings for the case study was not trivial, and maintaining similar mappings in the face of language evolution can also be a considerable problem.

To conclude this section we would like to discuss two positive aspects of this intermediate language case study. The first aspect is that the approach was made possible by the high expressive power of `XPM`: since `XPM` is capable of modeling such a high number of control flow patterns, we did not find BPMN's structures that were impossible to map into `XPM` structures. Nevertheless, not all of the mappings were trivial, and most `XPM_BPMN` models were more complex than the original BPMN models. The second positive aspect was making reified BPMN models available to inquire about the state of the processes. This simplified answering queries because requests and responses did not have to be translated into XPM_BPMN terms. However, this created the necessity of keeping the BPMN models updated with respect to the state of the XPM_BPMN models.

7.7.3 WS-BPEL

To further validate Cumbia with a well known language, we also did a case study based on WS-BPEL v2.0 [OAS05]. In this case, the engine that we produced was named Caffeine[11] and it implemented most of the language constructs[Muñ09, Rom07]. The elements not supported are mostly related to exception handling and they were left out of the case study because of time constraints. In spite of this, Caffeine is interesting because of the *language specific elements* that it required. These elements fulfill a very important role because they enable all the web-services communication. Figure 7.32 shows a schematic view of the elements and interfaces found in this engine.

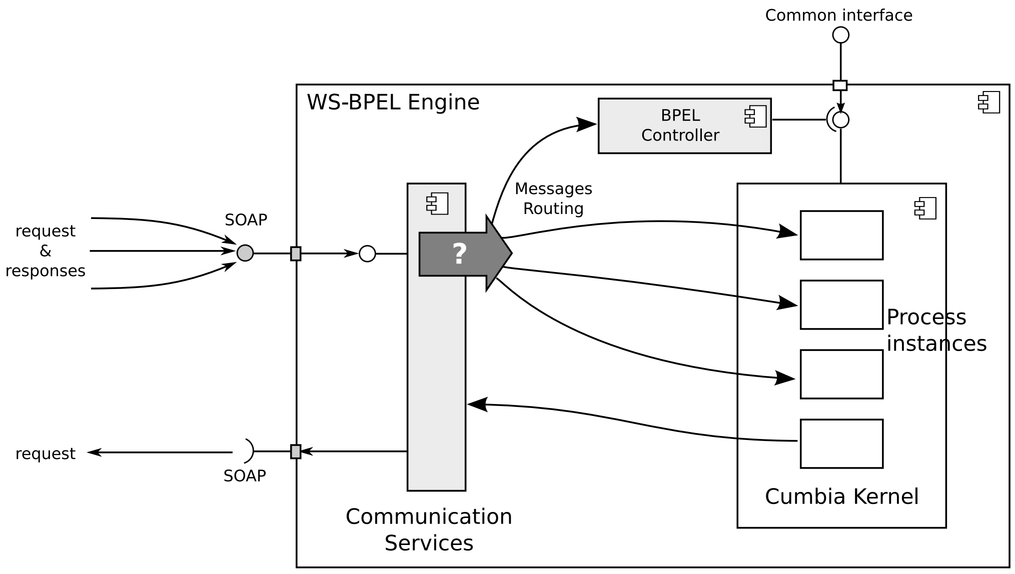


Figure 7.32 Schematic view of the WS-BPEL Engine (Caffeine)

Similarly to every Cumbia-based engine, the central component of Caffeine is the Cumbia Kernel which is configured with the WS-BPEL metamodel specification. This is complemented by a component that provides *communication services*. Through this component WS-BPEL instances can send requests to external web-services and receive their responses. The communication services component also supports asynchronous requests, although this entails the extra complexity of pairing responses to requests: this is achieved through a routing mechanism based on the correlation data included in the messages. Finally, the communication services component is also capable of identifying control messages and route them to the component called *BPEL Controller* which enacts them. This serves, for example, to load new process definitions, or to create new instances.

[11]Caffeine is the main result of the work of Manuel Muñoz at the Universidad de los Andes towards his MSc. degree. Manuel collaborated with Mario Sánchez in this project and his advisor was Jorge Villalobos. Daniel Romero, a former MSc. student at Universidad de los Andes also participated in Caffeine.

Introducing a component to handle all the communications was not absolutely necessary for the construction of Caffeine. An alternative would have been to implement all the communication behavior inside the metamodel. However, technical restrictions of the SOAP protocol and of the SOAP library used (AXIS) pushed us towards this solution.

In retrospective, the WS-BPEL case study did not gave us a lot of extra information about the expressiveness and extensibility of Cumbia and of the open objects. Nevertheless, because of the ubiquity of WS-BPEL and its usage as the base for other languages, it was a relevant scenario to implement. Furthermore, from the point of view of the engine architecture, Caffeine was an interesting case study that gave us valuable information about language specific elements and about patterns of interaction with external applications.

7.7.4 SCA

Service Component Architecture (SCA) is a set of specifications produced by OASIS which describe a model for building systems using a Service-Oriented Architecture (SOA) [OSO07, Bar07]. The idea behind SOA is that business functions can be provided as services. These can either be simple services, or service compositions (composites) formed by other services and by functions supported by existing applications. SCA provides the means to describe how services should be composed in composites, and also to describe how new services should be created. To enable the reuse of existing applications, SCA builds on open standards and supports binding components to systems implemented on technologies such as web-services, JMS, JCA, EJB, C++, and WS-BPEL.

SCA is the language used in another case study for Cumbia [12]. This language was selected because it is not a workflow specification language, but still has several similarities. Therefore, SCA served to test the applicability of Cumbia outside the context of workflow applications.

The principal elements defined in the SCA specification are shown in figure 7.33. An `SCA Domain` is a space in a container to deploy `composites`. Each `composite` assembles SCA elements in deployable logical groupings, and provides one or several `services`. A `composite` can also use `references` to declare the need for some externally defined components. The contents of a `composite` are `components` whose implementation is defined within the `composite`, and `wires`: `wires` are used to compose components by connecting `references` declared in one `component` to `services` provided by another one. A `component` can be an instance of another `composite`, or it can be implemented by another application (e.g. a Java, C++, or WS-BPEL application). Finally, the SCA model also defines `properties` both at the `composite` and at the `component` level: composite properties are defined when the `composite` is deployed; component level properties can be defined when the `composite` is built, or they can take their value from composite level properties.

Figure 7.34 uses SCA's graphical notation and depicts the structure of a new composite service. This "Account Service" is composed by two composites ("bigBank.Account" and "bibBank.AccountData") and the following three components: "Account Service Component", "Account Data", and "Account DataService

[12]The SCA scenario is the main result of the work of John Espitia at the Universidad de los Andes towards his MSc. degree [Esp09]. John worked closely with Mario Sánchez in this project and his advisor was Jorge Villalobos.

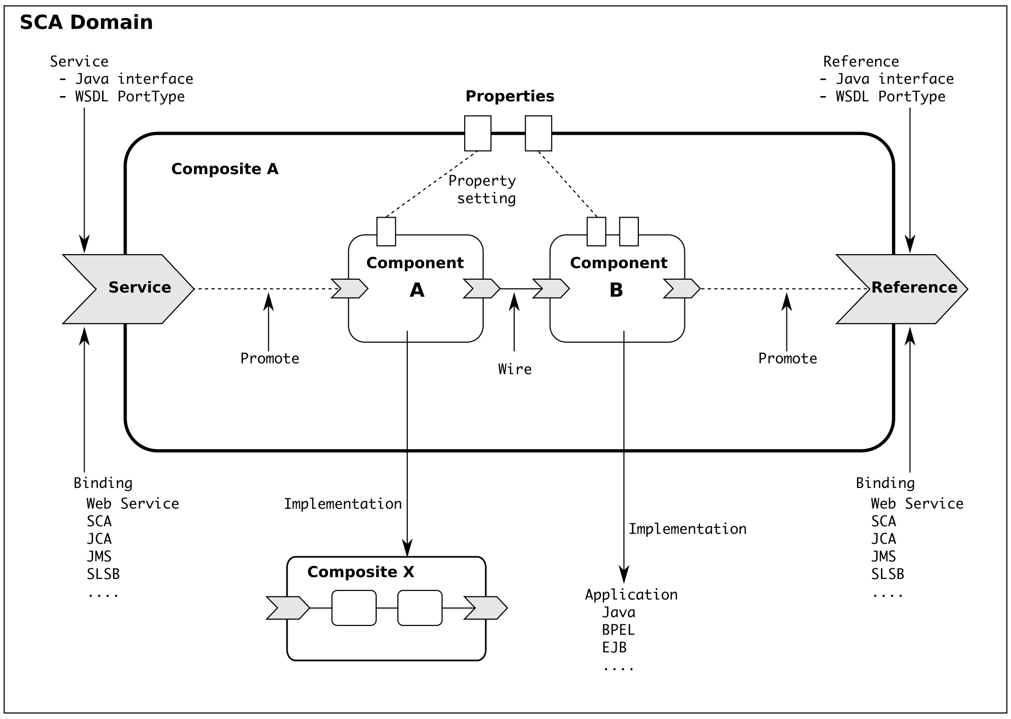


Figure 7.33 Elements of SCA, based on figures from [OSO07]

Component". The first two of these components are supported by external systems which are not visible in the figure. The third one, "Account DataService Component" is implemented by the composite "bigBank.AccountData". Finally, this application depends on an external service which is referred as "StockQuote Service".

The construction of the SCA container entailed the development of an SCA metamodel (see figure 7.35)[13], and wrapping the Cumbia Kernel as an SCA container. Subsequently we integrated this container with Oracle Business Activity Monitoring (Oracle BAM) [Ora09] to monitor the execution of the composites through dashboards. This integration was achieved using a JMS-based bridge and the open objects' coordination mechanisms. In particular, we installed actions on transitions and listeners for some events. Additionally, this case study also involved using other tools in the Cumbia platform, including the Cumbia Debugger and the Cumbia Test Framework.

This case studied the application of Cumbia in a domain that did not included workflows, and did so by implementing an existing language. The main conclusion that we can take from this is that Cumbia is also applicable in other contexts. A probable reason for this is that the metamodeling platform does not have a lot of restrictions and is not tied to particular workflow models and languages. Conversely, attempting a similar implementation on top of a platform such as OPERA would probably result in a flawed or limited implementation of SCA.

The SCA contained developed using Cumbia also benefits from characteristic **C2**: the elements of the specification were mapped to a Cumbia metamodel and

[13]Since the SCA scenario was developed the Cumbia platform has evolved in several ways, making obsolete the original implementation of the SCA metamodel. The image presented in this section corresponds to an updated, but not yet implemented, version of this metamodel, which takes advantage of all of the current features of Cumbia.

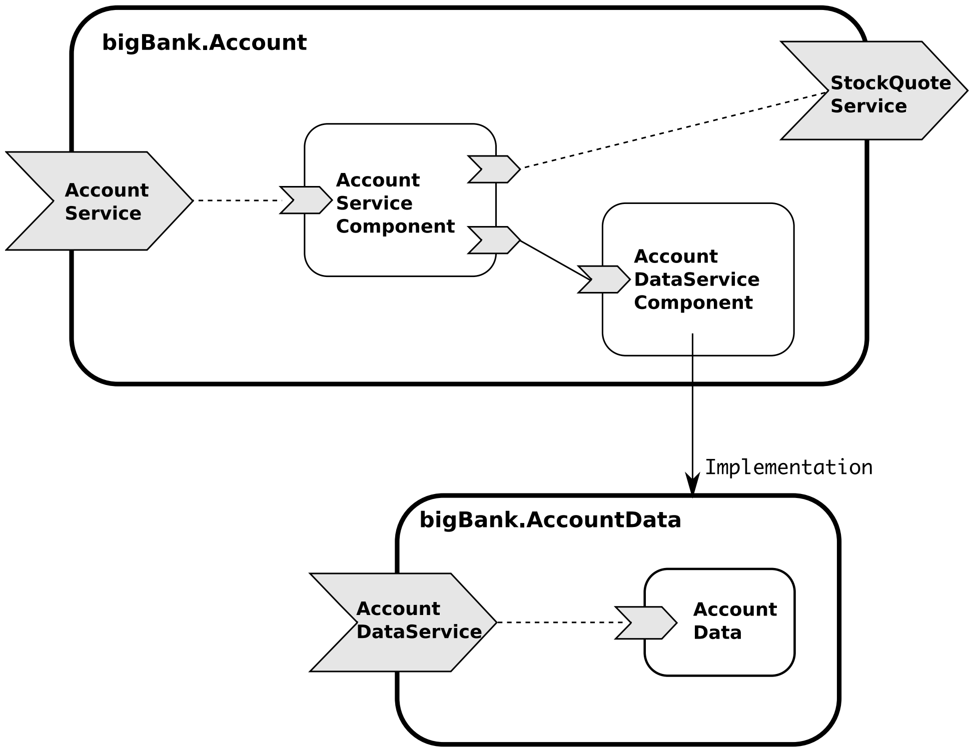


Figure 7.34 A sample SCA application. This figure can be found in [Bar07]

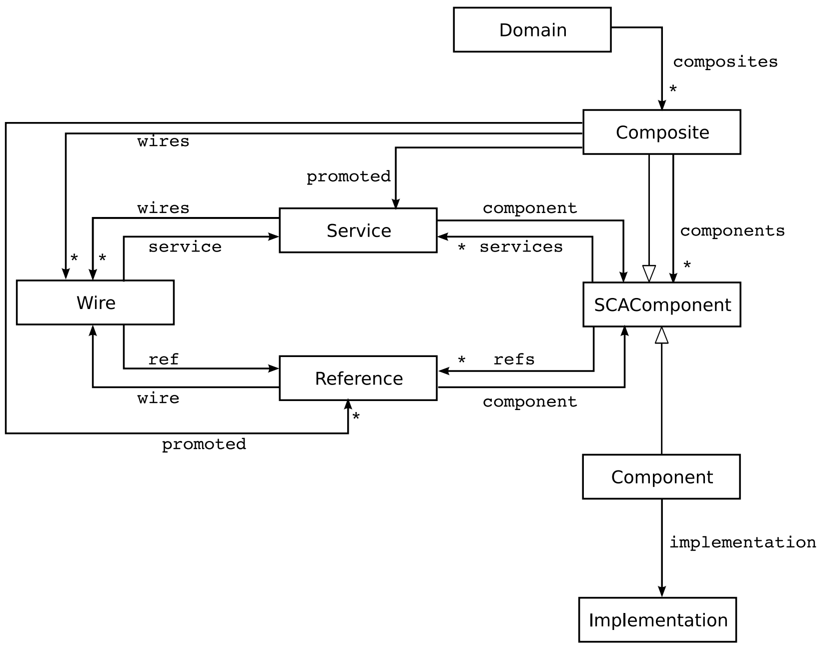


Figure 7.35 Metamodel for SCA (see note)

the implementation of its semantics was associated to elements of this metamodel. As a consequence of this, the container is flexible and it can take advantage of the extensibility mechanisms offered by open objects. This mechanisms can be used, for example, the create new bindings and implementation types by extending other elements already present in the metamodel.

With respect to all the discussed scenarios, in this one the characteristics **C4** and its advantages can be more clearly analyzed. In the first place, the platform was clearly reused: the only SCA specific elements were introduced either in the metamodel or as SCA-specific interfaces to the container. This not only reduced the effort to implement the container, but also allowed for the usage of tools like the debugger and the testing framework. On the other hand, the container that we developed is *open*, which makes it easy for external elements to observe its behavior, and to control it when necessary. On top of that, the facilities to observe events from outside of the container served to integrate it with the external monitoring application (Oracle BAM).

7.8 Review of the findings

In this chapter we have presented nine case studies built on top of Cumbia, which illustrate the application of the proposal to creating engines for a variety of workflow languages. For each language, we have presented its main characteristics and the most important details about its implementation. In those presentations we have focused on the aspects that made each case study different from the others, and the aspects that complicated the implementation and which required improvements to the platform. Furthermore, for some of the case studies we have also described some additional experiments that served to evaluate the extensibility of those Cumbia-based engines and applications.

The scenarios developed in this chapter were selected because they constitute an interesting subset of the spectrum of the workflow languages. The languages selected include some well known languages, as well as some ad hoc languages that we developed. Furthermore, we also included languages that target specific workflow domains. Finally, these languages are also based on different execution models, and thus they provided interesting scenarios to test the expressiveness of Cumbia and of the open objects.

Because of their different qualities, each case study has contributed to a different extent in demonstrating how the objective established in chapter 1, and revisited in section 7.1, was attained in the Cumbia platform. Furthermore, these case studies have served to illustrate the main characteristics of the platform and show how they contributed to achieving the objective. Table 7.1 summarizes how each scenario can be matched with each characteristic.

Characteristic **C1** makes reference to the lack of dependencies between the platform and specific workflow languages or type of execution model. Although we can not claim that Cumbia can support any workflow language, the case studies that we developed suggest that the objective of supporting a wide variety of workflow languages in a wide variety of domains was reasonably fulfilled. We implemented engines for representative languages and with profound differences in their structure, their notation, and their underlying execution model. These scenarios show that Cumbia supports several execution models, and does not depend on any of the traditional ones.

In all of the scenarios studied characteristic **C2** can be clearly seen. In each case, the language was modeled using metamodels, and its semantics was integrated into these metamodels. In some cases, such as in the WS-BPEL scenario, some elements had to be left out of the metamodel and be implemented as additional components in the engine. However, these elements were not central

Table 7.1 Match between case studies and characteristics of the platform

	C1	C2	C3	C4
`MiniBPMN`	+	++	+	
YAWL	+	+	+−	
Petri Nets	+	+		
PaperXpress	+	++	+	
IMS-LD	+	+	+−	+
XPM	+	+	+−	
BPMN	+	+		
WS-BPEL	+	+		
SCA	+	+		+

parts of the languages. Instead, they provided complementing services that did not determine the core parts of the language semantics. The other aspect of characteristic **C2**, namely aligning the evolution of languages and metamodels, was only studied in the case studies about `MiniBPMN` and PaperXpress. In those scenarios, the evolution of the languages can be easily mapped into the evolution of the metamodels.

Characteristic **C3** makes explicit the need for a platform where engines for new workflow languages could be easily changed. The case studies for `MiniBPMN` and PaperXpress illustrate that this goal was achieved in Cumbia. The changes introduced into these languages included changes to the behavior of existing elements, and the introduction of new elements. Furthermore, not all of the changes had to be introduced using the same means, because Cumbia offers different mechanisms to support different kinds of language extensions. The case studies for YAWL, IMS-LD, and XPM also contributed to demonstrating this objective, although in a different way. In these cases, the base languages were not extended, but they were complemented with additional languages. This mechanism supports a different kind of extensions that can be useful in other situations. Additionally, it promotes the reuse of engines.

With respect to characteristic **C4** there are three aspects to analyze. On the one hand, there is the fact that Cumbia is a reusable platform. All the scenarios studied are based on the same platform in spite of the profound differences between the languages. Furthermore, the advantage of reusing the platform is also evident in the permanent reuse of the Cumbia Debugger and the Cumbia Test Framework.

The second aspect is the open nature of the engines based on Cumbia. In particular, the scenario about IMS-LD showed that the interface of the Cumbia Kernel allows for complex manipulations of the models, even at run time. Without such a powerful interface, it would have been difficult to implement the dynamic adaptation functionalities required in the e-learning domain.

Finally, the third aspect has to do with the reuse of applications that interact with the new engines. Unfortunately, to test this aspect we were limited by the lack of reusable existing tools. Therefore, we can only present as evidence the reuse of our own tools, and the reuse of the Oracle BAM. The former include the Cumbia Debugger, the OO Editor, and the Cumbia Test Framework. The later, which was integrated to the SCA engine, was integrated using the event-based coordination mechanisms.

8 Conclusion

8.1 Summary and reflection

The popularization of workflows and their application to many different domains has given rise to a large number of workflow specification languages. Since each language has been designed following different goals, each one includes a different set of features and offers different advantages and disadvantages. Because of this, selecting the workflow specification language to use in a project requires a careful evaluation of the characteristics of the language, of the project, and of the domain where it is used.

Workflow specification languages can be classified as domain specific workflow languages (DSWfL) or as generic workflow languages (GWfL) depending on whether they target specific domains or not. Currently, the usage of GWfL is more extended, and the majority of the available tools target languages standardized by worldwide organizations and consortiums, such as BPMN or WS-BPEL. However, there are downsides in using GWfLs. The first one is that, in order to be generic, they manage concepts which are technical and are far from the concepts of the domains of application. Therefore, domain experts without a lot of technical knowledge cannot take advantage of many of the features present in these languages, which is paradoxical since they were originally intended to be the main beneficiaries of workflow technologies. The second aspect is the limited flexibility offered by those languages. Since the specifications are fixed, and since many implementations are proprietary, it is very difficult to adapt these languages to offer support for new requirements that were not considered when they were designed.

On the other hand, domain specific workflow languages explicitly handle concepts taken from the domains of the application, and are expected to handle ad hoc requirements. The problems with those languages arise from the high costs associated to their development and maintenance. This has happened because existing engines are tightly coupled to the languages that they support, and because there is relative lack of tools to support the definition and execution of DSWfLs. As a result, the engines for new workflow languages are developed almost from scratch, and requires the re-development of all the necessary associated tools.

Given this context, the goal of this dissertation was to develop the means to support the implementation of new workflow languages, and, at the same time, to favor and support flexibility at the language level. The proposal to achieve this goal was a novel architecture to build workflow engines, based on metamodels and on the modularization of workflow definitions. In this architecture, engines are built on top of a metamodeling platform and an execution kernel: for each workflow language, a metamodel and an engine are developed, but this development does not entail as much effort as it is normally required.

The proposed architecture introduces the possibility of modularizing workflows according to concerns, and using concern specific languages to describe each one of those. Since these languages target narrow facets of the workflows, they can be very expressive and manage concepts that are familiar to domain experts. At the same time, the number of elements managed in these languages is small, and this reduces the effort required to implement and maintain each language. On top of that, the proposed platform offers elements and tools that can be reused in many languages and that also reduce the effort to support new languages. All these facts favor the creation of new concern specific languages as an alternative to reusing complex and monolithic generic workflow languages that manage concepts which are far from specific domains.

Besides facilitating the creation of new languages, the proposed approach also favors language adaptability, extensibility, and reuse. Adaptability is favored because implementing the languages directly in metamodels aligns languages and their implementations in ways that are not achievable in other approaches. In this way, changes to the languages' elements and structures are easy to map into changes to the metamodels. Extensibility is favored because the underlying metamodeling platform offers extensible constructs and encapsulates in each element as much as possible of its own behavior. Thus, the impact of extending the elements of a language is minimal. In addition to this, applications can also be extended by introducing new languages that are externally composed to the existing ones. With respect to reuse, modularizing the languages increases their chances of been used in several applications, either with a few changes or with no changes at all.

The proposed architecture also introduces elements that favor the creation of open engines that can be easily integrated with external tools. At the base of this, there are very open interfaces that can be used to observe and control the execution of the models, and even to modify their structure at run time. The widespread usage of events is also helpful as every event generated in an engine can be easily observed by external applications. Therefore, it is easy to develop external applications to monitor these engines where their run time behavior is described as reactions to events. Finally, in the proposed architecture all the engines share a number of components. External applications that use the interfaces proposed by the reusable platform have the potential to be used with multiple engines without requiring substantial changes.

All the above has been implemented in the Cumbia platform, on top of a notion that we developed which is called open object. Open objects can be considered the core of the metamodeling platform: they are the modeling abstraction to describe the elements in the metamodel, which includes their run time state and behavior. To describe this behavior, open objects rely on coordination mechanisms like events and method invocations, and on externalized and reified state machines. These coordination mechanisms are not limited to be used just within a model. They can also be used from outside each model to get information about its execution and to control it. In the proposed platform, all of this is used to coordinate the execution of multiple models representing different concerns of a workflow.

The metamodels for various concerns are normally developed independently. Because of this, external descriptions are required to establish the relations between models using a language called CCL. With CCL it is possible to describe how to weave model instances using the coordination mechanisms offered by the open objects (events and method invocation).

All the aforementioned elements of the proposal contribute to achieving the intended goals and support the development of engines for new workflow languages, while making them adaptable and extensible. It is evident that many of the functionalities commonly required in workflow engines do not have to be re-implemented for each language, as they are provided by the platform itself. Furthermore, complementary tools can be reused as long as they target the platform and do not depend on language specific features. However, it is not easy to argue whether the resulting engine implementations are more or less *complex* to produce than comparable implementations based on more traditional approaches. Compared to those, developing metamodels based on open objects requires other skills, such as familiarity with the open objects, and the capacity to design highly concurrent models. In this particular aspect we have encountered the most problems while developing our case studies. While the structure of the metamodels and the *internal* behavior of each element are easy to design, it can be very complex to design their interaction. In some cases, it required a lot of effort to stabilize the metamodels, i.e. achieve the correct synchronization of the elements and eliminate problems such as deadlocks or race conditions. We consider that some of these problems can be mitigated with adequate tool support, and thus we have included the development of new and more powerful tools in the proposed future work.

To guide the usage of the numerous elements of the proposal, we have described a generic development process. This process describes the steps to create new applications based on Cumbia, and organizes them according to the products that they require and they produce. We have also identified the tools that support each step, and the skills that participants need to successfully perform each one of the tasks.

Cumbia was used in a number of case studies that we presented in chapter 7. In each one of those, a different workflow language was implemented, and several experiments were performed, including an implementation of an engine. The languages studied in those case studies were a representative set of both generic workflow languages, and domain specific workflow languages. The domains studied included those of business process management, e-learning, and collaborative writing. Furthermore, we also performed some experiments using SCA, a specification that serves to describe services in service oriented architectures.

8.2 Conclusions

In this section we summarize and discuss the main contributions of this dissertation.

A novel architecture for workflow management systems

In this dissertation we presented the elements of a novel architecture to build workflow management systems. In this architecture, languages can vary but the implementation of functionalities common to most workflow languages is always reused. This strategy contributes to diminishing the costs associated to supporting new languages, and facilitates the reuse of external complementary tools.

The fundamental elements in this architecture are the metamodeling platform based on open objects, and a kernel for the execution of models – the Cumbia Kernel. The metamodeling platform provides the elements to create metamodels to implement workflow languages. Models conformant to those metamodels, and thus representing specific workflows, are executed by the Cumbia Kernel. This means that the kernel follows the semantics of the language to update the state of the models and to control the interaction with external elements. The proposed architecture is complemented by the Cumbia Weaver. This component is in charge of interpreting CCL programs and materializing the relations between models that are necessary to coordinate their execution.

Finally, it is important to highlight the central role of open objects in this architecture. Firstly, open objects provide the elements to describe the metamodels, including the structure of valid models, the behavior of its elements, and the interactions between those elements. Furthermore, open objects offer various extension mechanisms which contribute to the evolution of the languages. The interaction mechanisms of the open objects are also used by the Cumbia Weaver to realize the coordination between models. Finally, the open objects expose as much as possible of their state in order to facilitate the development of open engines, and the integration with external applications.

A novel approach for workflow modularization

This dissertation explored a multimodeling approach to modularize workflow definition. In this approach, workflow specifications are decomposed among a number of concerns, and each part is specified using a concern specific language. This approach brings advantages both from the point of view of the implementors of the languages and from the point of view of its users. In the first case, this is explained because concern specific workflow languages tend to be simpler and smaller than fully fledged workflow languages. Therefore, both the development and maintenance of each one of those languages is simplified. From the point of view of the users, which in many cases are domain experts, these languages are easier to use and to learn, because they manage concepts that are closer to their domains of expertise.

This workflow modularization approach is implemented on top of the open objects based metamodeling platform. For each concern specific language, one metamodel has to be developed. Later on, CCL and the open object coordination mechanisms are used to coordinate the execution of the concern specific models and thus reconstruct the semantics of the modularized languages.

Open Objects, a base element for executable modeling

In this dissertation we developed the notion of open objects and used it as the main building block for metamodels. Since we were interested in having executable models, the specification of the metamodels and of the element types had to fulfill special requirements. Firstly, each element type needed a way to store the run time state of its instances. Secondly, the metamodel had to define the structure of conformant models. And thirdly, each element type had to define the behavior of its instances with respect to other elements and to the execution environment.

These requirements were all fulfilled by open objects. The attributes of the entities, and the states in the state machines, provide the means to store and structure the run time state of element and model instances. Each open object also defines how its instances can relate to other elements in the metamodels, thus restricting the structure of valid models. Finally, the behavior implemented in the methods of the entities and in the actions, together with the state machines and the event passing mechanisms, provide the means to define the behavior of metamodels. The previously mentioned characteristic also makes open objects very open, and thus it is easy to observe their execution, receive notifications about their state changes, or extend their behavior with new actions.

Finally, open objects' usage is not exclusive to workflows: they can be used in other contexts, as was demonstrated in the SCA case study.

The Cumbia platform

We implemented all the elements proposed in the approach in a platform called Cumbia. This platform supports the definition and implementation of workflow languages using metamodels based on open objects. It also supports the execution of models conformant to those metamodels, and supports the run time coordination of multiple models. Furthermore, the Cumbia platform exposes powerful interfaces to control and monitor the execution of the models.

The core of Cumbia has three main elements. Firstly, a metamodeling platform and a framework for open objects that serves to design and implement metamodels and models. Secondly, a component called the Cumbia Kernel, where models can be executed accordingly to the semantics of each language. And thirdly, a component that weaves the models so their executions can be coordinated, the Cumbia Weaver. Additionally, we have also implemented complementary tools that aid in the development of applications based on Cumbia. These tools include a debugger, an editor to structure the metamodels, and a framework to create testing environments. To guide the usage of all these components and tools we have suggested a development process that describes the steps that participants with different roles have to perform to create new applications based on Cumbia.

From the technological point of view, the Cumbia platform uses Java and XML: metamodels and models are described using XML files, while the behavior of open objects' entities is implemented in Java classes.

Validation of the Cumbia platform

The Cumbia platform was validated with nine case studies that present relevant characteristics commonly shared by workflow languages. The languages, specifications and models studied in these cases comprised both generic workflow languages and domain specific workflow languages. Among the domain specific languages we also included SCA, a specification to design service oriented architectures that shares many characteristics with traditional workflow languages. Furthermore, the languages selected for the case studies used various types of execution models (based on Petri nets, transition based systems, extended workflow nets), thus validating that our platform is not tied to a particular type of execution model.

In each case study, we built an engine for one of the languages using the elements provided in the Cumbia platform. Furthermore, in several of the case studies we performed additional experiments that included modifications and extensions to the languages, the development of new complementary languages, and the composition with existing languages.

With these case studies we concluded that Cumbia fulfills the objectives of this dissertation: it is a platform where engines for workflow languages can be implemented while favoring language extensibility and reusability.

8.3 Future work

This section presents a number of directions for future research related to Cumbia, as well as some possible improvements to its current implementation.

8.3.1 Possible improvements and additions for Cumbia

Verification of metamodels

The current implementation of the Cumbia platform is capable of detecting syntactic problems in the metamodels. For example, it can detect if an element's state machine depends on an event that it will never receive because it is not declared by any element. Similar problems can be detected in models by analyzing its elements and matching them to the metamodels. These verifications are very useful to metamodel developers as they reveal errors such as simple typos and omissions.

Unfortunately, the problems faced by metamodel developers are frequently much more complex than syntactic errors. For example, it is relatively easy to introduce deadlocks, i.e. two elements mutually waiting an event from the other to generate their event. It is not as easy to detect these situations, especially when the deadlock involves long chains of elements. Other potential problems are those where elements can reach inconsistent states because events can be received in an unexpected order.

In complex concurrent contexts such as Cumbia, test-based approaches cannot rule out the possibility that models conformant to some metamodel are ever going to reach inconsistent states. These approaches provide insights about the quality of the design, and can uncover problems, but they do not serve to prove whether a metamodel is flawless. Therefore, it is necessary to develop or adapt tools and methods to perform analytical verification of metamodels. Eventually, these tools will have to focus on three aspects. Firstly, detecting if some generic problematic situations can occur (e.g. deadlocks). Secondly, detecting if some undesirable states are reachable. Since the specific states depend on the metamodel, this aspect would also require some mechanisms to describe the undesirable states. Finally, the same tool that detects the reachability of undesirable states would also be used to decide if desirable states (final or intermediate) are *always* reachable.

Further changes to the platform and to the open objects model

This dissertation presented an open objects model that differs in many aspects from its first version. Many changes have been introduced gradually into the platform, motivated by new requirements and by the findings in the case studies developed. At some points, we have even introduced features that we have retracted later on. We anticipate that the platform will continue to evolve in the near future as new experiments are performed, but we expect the changes to be less profound as the platform matures.

Given the necessity that we have seen for verification mechanisms, and given the current complexity of the platform, we believe that it is desirable to explore mechanisms to simplify it. This does not necessarily has to occur in the main trunk of the platform. For example, we could create a parallel version geared specifically towards verification and reasoning. In this *branch for verification* we would sacrifice expressiveness to facilitate the analysis and validation of the core aspects of a metamodel design.

Another idea for the open objects model that we should explore in the near future is the inclusion of guards in transitions. Currently, transitions in state machines are triggered whenever an event of the right type, and emitted by the right element, is received. The problems related to deadlocks and unreachable states that we faced during the development of the case studies were mainly caused by events received and processed at the wrong moment. Associating guards to transitions seems like a viable alternative to reduce these problems. Additionally, putting guards in transitions also seems useful to facilitate the validation (via testing) of metamodel designs, as they can fulfill a function similar to that of assertions in Java and similar languages.

Finally, another interesting extension to the open objects model could involve the declaration of expressions about the intended behavior of some metamodels or about their constraints. The objective behind these could be to prevent the introduction of errors by metamodel extensions, or by metamodel compositions.

Relations between metamodels

In this dissertation we have presented all the mechanisms that Cumbia offers to first modularize the languages, and then to allow the coordinated execution of definitions written with those modularized languages. The modularization of the languages is based on the identification of concerns, and it is reflected in a collection of concern specific metamodels that are largely independent. Nevertheless, not in all situations it is possible, or desirable, to achieve such a clean separation between concerns and between metamodels. For example, if concerns are not completely orthogonal, it may be desirable to see the *same* element in several concerns and several metamodels. The current approach totally decouples the models and leaves to CCL programs the responsibility of guaranteeing consistency. Also, a mechanism is missing to ensure that the elements in one metamodel offer the composition hooks that the elements in another metamodel needs.

Because of these reasons, we consider that future developments of the Cumbia platform should introduce the possibility of having explicit dependencies between metamodels. We currently envision two kinds of dependencies. In the first and simpler, one metamodel will *import* an element from another metamodel, and use it as if it was locally defined. In the second one, an *interface* element will be described and the rest of the elements will be developed against that interface. At a later time, the interface will be bound to a concrete element defined in another metamodel. At the model level, these two approaches will probably require the usage of placeholders, and CCL will have to be enriched to replace the placeholders with references to the real elements, at weaving time.

Other improvements to CCL

CCL is a low level language based on low level coordination primitives. Therefore, to write CCL programs it is necessary to have a strong understanding of the structure of the open objects involved in the composition. In particular, the author of a CCL program has to know all the details about the structure of the relevant state machines, about the elements behavior, and about the events expected and generated. The developer needs this knowledge to ensure that the composition is not going to break the models: without this information, he simply cannot write CCL programs. These requirements restrict the usage of CCL to expert users. A further problem is the absence of relations between metamodels, which make it impossible to define what a semantically valid composition is. Thus, only syntactic verifications are made on CCL programs, and semantic errors cannot be caught.

To address these issues, we will have to extend the Cumbia architecture with the means to describe valid compositions at the metamodel level [RSV11]. We are currently doing so by implementing a solution that reflects the approaches used to describe model weaving in AMW [FBJ$^+$05]. In this solution, composition models are used to establish the relations between elements in different metamodels that can be considered valid. The descriptions of those relations also include information to map relations to CCL programs. By doing so, when the relations are instantiated at the model instance level, a corresponding CCL program can be generated and executed. This is also inspired in the approach of Melusine and Focas, where *i*) a composition domain is established to relate existing domains; *ii*) composition models conformant to the composition domain are used to establish relationships between application models; *iii*) and at last the execution of the application models is synchronized using the information described in the composition model [EIV05, PE08].

The strategy that we are developing has four key points that make it a solution to our problems. The first one is that the valid relations can be of a high level and thus be usable by non-technical users. Furthermore, this approach enables the construction of *composition editors* that provide more assistance to users than the CCL editor that we currently have. A third key point of this approach is allowing the detection of semantic problems in the compositions. Since compositions are limited to the relations established between the metamodels, semantic problems can only be introduced if the relation definitions have errors. Finally, it is important that this solution is based on CCL and the Cumbia Weaver. Because of this particular detail, defining the relations between two or more metamodel requires an effort similar to the one was previously required to write a CCL program. The difference is that, in this case, this work is done by a metamodel developer, while previously it was supposed to be a task of a model developer.

Other improvements to CCL will serve to handle finer grained events, instead of the coarse ones that are currently supported. As an example, we expect to be able to describe that a given block of weaving code has to be executed when a certain kind of element is dynamically created into a model. The need for such improvements comes, in general, from scenarios that require dynamic adaptation, and specifically, from IMS-LD.

8.3.2 Concrete syntax

In this dissertation we have focused on the behavioral aspect of the languages and we have only provided support for the definition of their abstract syntaxes. Nevertheless, the concrete syntaxes are essential to many applications and are naturally the next step to tackle in Cumbia. Some of the metamodeling platforms that currently support the definition of language notations include MetaCase [LKT04], GME [Ins08], and Fujaba [GZ05]. However, they cannot be used with Cumbia because they are incompatible with the open objects: their metamodeling stack has a fixed height and it is based on a fixed meta-metamodel. As a result, it is not possible to replicate, inside those platforms, the metamodeling structure that we have presented throughout this dissertation.

To support the definition and usage of notations in Cumbia we have identified two alternatives. The first one is to develop the necessary tools and theories from the ground up and make them compatible with open objects since their inception. However, to support advanced notations (including graphical ones) and offer the features found in most DSL platforms a big development effort is going to be required. The second alternative is to adapt one of the existing platforms. The difficulty of this approach will depend on the specific characteristics of the selected platform, and on the level of integration sought with the rest of the Cumbia platform.

8.3.3 Management of evolution

Throughout this dissertation we have discussed the topics of language and metamodel evolution. A related topic that we have not addressed is that of model and metamodel co-evolution: since metamodels representing languages are going to evolve, it is relevant to consider how this is going to affect the models conformant to them. Currently, we have avoided consistency problems by using a version-based system. However, this means that models conformant to old metamodels cannot be used with the newer versions unless they are manually updated. Sometimes, this only requires changing an attribute in their specification. This simple strategy eliminates some evolution problems because creating a new version of a metamodel does not affect existing models. On the other hand, this strategy makes it difficult to upgrade old models or fix bugs in the metamodel implementations. Therefore, this is an interesting topic to study with Cumbia, and it seems like it would be possible to apply techniques similar to those presented in

COPE [HBJ09]. The reason for this is that the *coupled evolution* of models and metamodels can happen with types of changes that are known in advance.

A
Interfaces of the Open Objects

This appendix lists the main Java interfaces for model elements in Cumbia. The framework provides default implementations for these interfaces, and thus metamodel developers do not have to worry about implementing them. The services described in these interfaces are heavily used by the Cumbia Kernel, and they can also be used in specific metamodels.

A.1 INavigable

Listing A.1: INavigable

```java
/**
 * This interface defines the methods implemented by elements in navigable
     models
 */
public interface INavigable
{
    // ------------------------------------------------------------------
    // Methods
    // ------------------------------------------------------------------

    /**
     * Follows a relation and obtains the referenced element
     * @param relationName The name of the relation to follow
     * @return Returns the element found, or null if the relation is empty
     * @throws InvalidRelationException This exception is raised if the name
         of the relation does not exist
     * @throws InvalidCardinalityException This exception is raised if the
         cardinality of the relation is more than 1
     */
    INavigable getElement( String relationName ) throws
        InvalidRelationException, InvalidCardinalityException;

    /**
     * Follows a relation and builds a collection with the elements
         currently referenced in the relation. If the relation is a
         sequence, the
     * collection returned is implemented with a List.
     * @param relationName The name of the relation to follow
     * @return Returns a collection with referenced elements. If no elements
         are referenced, the return is an empty collection. If the
     *         cardinality of the relation is one, a collection is still returned.
     * @throws InvalidRelationException This exception is raised if the name of the relation does not exist
     */
    Collection<INavigable> getElements( String relationName ) throws InvalidRelationException;

    /**
     * Follows a sequential relation and obtains the element referenced in a given position.
     * @param relationName The name of the relation
     * @param position The position in the sequence of references sought.
     * @return Returns the referenced element
     * @throws InvalidRelationException This exception is raised if the name of the relation does not exist
     * @throws InvalidTypeException This exception is raised if the relation is not implemented with a sequence
     * @throws InvalidKeyOrPositionException This exception is raised if the position indicated does not exist in the sequence
     */
    INavigable getElement( String relationName, int position ) throws InvalidRelationException, InvalidTypeException, InvalidKeyOrPositionException;

    /**
     * Follows a map relation and obtains the element referenced with a given key.
     * @param relationName The name of the relation
     * @param key The key in the map of references sought.
     * @return Returns the referenced element
     * @throws InvalidRelationException This exception is raised if the name of the relation does not exist
     * @throws InvalidTypeException This exception is raised if the relation is not implemented with a map
     * @throws InvalidKeyOrPositionException This exception is raised if the key indicated does not exist in the sequence
     */
    INavigable getElement( String relationName, String key ) throws InvalidRelationException, InvalidTypeException, InvalidKeyOrPositionException;

    /**
     * Sets the element referenced in a relation. If another element was referenced before, that reference is lost.
     * @param relationName The name of the relation
     * @param referenced The object that will now be referenced
     * @throws InvalidRelationException This exception is raised if the name of the relation does not exist
     * @throws InvalidCardinalityException This exception is raised if the cardinality of the relation is more than 1
     */
    void setElement( String relationName, INavigable referenced ) throws InvalidRelationException, InvalidCardinalityException;

    /**
     * Adds a reference to a relation implemented with a sequence
     * @param relationName The name of the relation
     * @param referenced The new object to be referenced
     * @return Returns the position where the element was stored in the sequence
     * @throws InvalidRelationException This exception is raised if the name of the relation does not exist
     * @throws InvalidTypeException This exception is raised if the relation is not implemented with a sequence
     */
    int addElement( String relationName, INavigable referenced ) throws InvalidRelationException, InvalidTypeException;

    /**
```

```
     * Adds a reference to a relation implemented with a map. If there was
         already a reference stored with the same key, the old reference
     * is lost.
     * @param relationName The name of the relation
     * @param key The key in the map of references
     * @param referenced The new object to be referenced
     * @throws InvalidRelationException This exception is raised if the name
          of the relation does not exist
     * @throws InvalidTypeException This exception is raised if the relation
          is not implemented with a map
     */
    void addElement( String relationName, String key, INavigable referenced
        ) throws InvalidRelationException, InvalidTypeException;

    /**
     * Adds a list of references to a relation implemented with a sequence.
         The order of the references is maintained, and the new
     * references are added at the end of the sequence. If an element is
         duplicated, it is added twice.
     * @param relationName The name of the relation
     * @param referenced The collection of objects to be referenced
     * @return Returns the number of references added to the sequence
     * @throws InvalidRelationException This exception is raised if the name
          of the relation does not exist
     * @throws InvalidTypeException This exception is raised if the relation
          is not implemented with a sequence
     */
    int addAll( String relationName, List<INavigable> referenced ) throws
        InvalidRelationException, InvalidTypeException;

    /**
     * Adds a map of references to a relation implemented with a map. The
         new references always replace existing references if they are
     * stored with the same key.
     * @param relationName The name of the relation
     * @param referenced The collection of objects to be referenced
     * @return Returns the number of references added to the map
     * @throws InvalidRelationException This exception is raised if the name
          of the relation does not exist
     * @throws InvalidTypeException This exception is raised if the relation
          is not implemented with a map
     */
    int addAll( String relationName, Map<String, INavigable> referenced )
        throws InvalidRelationException, InvalidTypeException;

    /**
     * Removes the element referenced in a relation. The relation has to be
         simple. If there is no referenced element nothing happens.
     * @param relationName The name of the relation
     * @return Returns the reference that was removed
     * @throws InvalidRelationException This exception is raised if the name
          of the relation does not exist
     * @throws InvalidCardinalityException This exception is raised if the
         relation is not simple.
     */
    INavigable removeElement( String relationName ) throws
        InvalidRelationException, InvalidCardinalityException;

    /**
     * Removes an element referenced in a relation, given its position. The
         relation has to be a sequence.
     * @param relationName The name of the relation
     * @param position The position of the reference in the sequence
     * @return Returns the reference that was removed
     * @throws InvalidRelationException This exception is raised if the name
          of the relation does not exist
     * @throws InvalidTypeException This exception is raised if the relation
          is not a sequence.
     * @throws InvalidKeyOrPositionException This exception is raised if the
          position is inexistent in the sequence
     */
```

```java
    INavigable removeElement( String relationName, int position ) throws
        InvalidRelationException, InvalidTypeException,
        InvalidKeyOrPositionException;

    /**
     * Removes an element referenced in a relation, given its key. The
         relation has to be a map.
     * @param relationName The name of the relation
     * @param key The position of the reference in the map
     * @return Returns the reference that was removed
     * @throws InvalidRelationException This exception is raised if the name
          of the relation does not exist
     * @throws InvalidTypeException This exception is raised if the relation
          is not a map.
     * @throws InvalidKeyOrPositionException This exception is raised if the
          key is not found the map
     */
    INavigable removeElement( String relationName, String key ) throws
        InvalidRelationException, InvalidTypeException,
        InvalidKeyOrPositionException;

    /**
     * Removes an element referenced in a relation, given the object to
         remove.
     * @param relationName The name of the relation
     * @param referencedElement The element to remove from the relation
     * @throws InvalidRelationException This exception is raised if the name
          of the relation does not exist
     * @throws InvalidCardinalityException This exception is raised if the
         relation is not multiple.
     * @throws InvalidKeyOrPositionException This exception is raised if the
          element is not part of the relation
     */
    void removeElement( String relationName, Object referencedElement )
        throws InvalidRelationException, InvalidCardinalityException,
        InvalidKeyOrPositionException;

    /**
     * Removes all the references associated to a relation. This can be
         applied either to multiple or to single relations.
     * @param relationName The name of the relation
     * @return Returns the number of references removed
     * @throws InvalidRelationException This exception is raised if the name
          of the relation does not exist
     */
    int removeAll( String relationName ) throws InvalidRelationException;

    /**
     * This method retrieves the information about the relation
     * @param relationName The name of the relation
     * @return The information about the relation
     * @throws InvalidRelationException This exception is raised if the name
          of the relation does not exist
     */
    RelationInfo getRelationInfo( String relationName ) throws
        InvalidRelationException;

    /**
     * This method adds a new relation to a navigable element
     * @param newRelation The information about the new relation
     * @throws InvalidRelationException This exception is raised if there is
          an existing relation with the same name
     * @throws InvalidRelationInformationException This exception is raised
         if the information provided is not valid
     */
    void addRelation( RelationInfo newRelation ) throws
        InvalidRelationException, InvalidRelationInformationException;

    /**
     * This method removes a relation. If there were references associated
         to this relation, those are lost.
     * @param relationName The name of the relation
     * @throws InvalidRelationException This exception is raised if there isn’t a relation with the name provided
     */
    void removeRelation( String relationName ) throws InvalidRelationException;

    /**
     * Queries the number of references currently contained in a relation.
     * @param relationName The name of the relation.
     * @return Returns the number of elements: for simple relations, the number can be 0 or 1; for multiple relations it depends on the name
     *         of ’used’ positions (or keys). If there are keys with non−binded references those should not be counted.
     * @throws InvalidRelationException This exception is raised if there isn’t a relation with the name provided
     */
    int getCurrentRelationCardinality( String relationName ) throws InvalidRelationException;

    /**
     * Returns a collection with the names of all the relations in the element.
     * @return A collection with relation names. This collection can be empty. Relations should be included, even if they don’t have
     *         associated references.
     */
    Collection<String> getRelationNames( );
}
```

A.2 IModelElement

Listing A.2: IModelElement

```java
public interface IModelElement extends INavigable
{
    /**
     * Returns the name of the element
     *
     * @return elementName
     */
    String getElementName( );

    /**
     * Returns the type of the element
     *
     * @return typeName
     */
    String getTypeName( );

    /**
     * Returns the memory of the element
     *
     * @return memory
     */
    Memory getMemory( );

    /**
     * Returns the instance where the element is used. This method has been
         restricted to return instances of KernelModelInstance
     *
     * @return modelInstance
     */
    IModelInstance getModelInstance( );

    /**
     * Changes the model instance referenced in the element
     *
     * @param modelInstance the new model instance
     */
    void setModelInstance( IModelInstance modelInstance );

    /**
     * Returns the information about the type of the element. The
         information is taken from the metamodel.
     * @return typeInformation
     */
    ITypeInformation getTypeInformation( );
}
```

A.3 IOOKernelElement

Listing A.3: IOOKernelElement

```java
/**
 * This is the interface that defines all the methods offered by a Kernel
     Element, that is, an element that is used in a model.
 */
public interface IOOKernelElement extends IModelElement
{

    /**
     * Returns the instance where the element is used. This method has been
         restricted to return instances of KernelModelInstance
     *
     * @return modelInstance
     */
    OOModelInstance getModelInstance( );

    /**
     * Returns the information about the type of the element. The information is taken from the metamodel.
     * @return typeInformation
     */
    OOTypeInformation getTypeInformation( );

    /**
     * Returns the events manager that handles the event generators of this element
     *
     * @return eventsManager
     */
    EventsManager getEventsManager( );

    /**
     * This method registers in the EventsManager of the element, all the events that are generated by this entity.
     * @param generatedEvents This is a list of the events declared as generated by the entity
     */
    void registerGeneratedEvents( List<Integer> generatedEvents );

    /**
     * This method returns the queue where events directed to this element can be delivered. If the element doesn't expects messages, this method returns null.
     *
     * @return returns the event queue of the element or null if the element should not receive events
     */
    IEventListener getEventQueue( );

    /**
     * If it is necessary, this method starts processing the events received by the element.
     */
    void startProcessingEvents( );

    /**
     * This method should subscribe this element to the events it requires to control its execution. For achieving this, this method should use the method
     * registerForEvent(EventInformation information), which implements all the steps necessary to create the subscription.
     */
    void subscribeToEvents( );

    /**
     * Registers this object's eventQueue as a listener for the given event
     *
     * @param information Information about the event that the event queue expects.
     */
    void registerForEvent( EventInformation information );

    /**
     * This method should subscribe this element to the events it requires to control its execution and can be generated by the new element. For achieving this, this method
     * should use the method registerForEventOfNewElement(EventInformation information, KernelElement newGenerator), which implements all the steps necessary to create the
     * subscription.
     */
    void subscribeToEventsOfNewElement( IOOKernelElement newGenerator );

    /**
```

```
     * If necessary, this method registers this object as a listener of an
         event generated by the new element. If the new element does not
         fulfills the role described in the
     * event, this method has no effect.
     *
     * @param information Information about the event that the event queue
         expects.
     * @param newGenerator The new element that will generate the event.
     */
    void registerForEventOfNewElement( EventInformation information,
        IOOKernelElement newGenerator );

    /**
     * This method disconnects the element from all the elements that
         generate events that its event queue expects.
     *
     * @param generators A list of elements that generate events that this
         element's event queue expects.
     */
    void disconnectFromAllGenerators( List<IOOKernelElement> generators );

    /**
     * This method disconnects the element from all its listeners. This is
         achieved by disconnecting each of the event generators that are
         managed by the EventsManager.
     */
    void disconnectFromAllListeners( );

    /**
     * Initializes the element after it has been created and the entire
         model has been connected
     */
    void initialize( );
}
```

A.4 IOpenObject

Listing A.4: IOpenObject

```
/**
 * This is the interface of an object. The main difference between an open
     object and a plain kernel element is the existence of a state
 * machine.
 */
public interface IOpenObject extends IOOKernelElement
{

    /**
     * Returns the state machine
     *
     * @return stateMachine
     */
    IStateMachine getStateMachine( );

    /**
     * Sets the state machine of the open object
     *
     * @param stateMachine State machine that has to be associated to the
         open object
     */
    void setStateMachine( IStateMachine stateMachine );

}
```

B
Metamodel and Model Specification Schemas

B.1 Metamodel definition schema

Listing B.1: Metamodel definition schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
    qualified">
  <xs:element name="metamodel">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="state-machine-
            reference"
          type="sm-reference_type"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="type" type="
            type_type"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="extended-type"
          type="extended-type_type"/>
        <xs:element minOccurs="1" maxOccurs="1" name="runtime" type="
            runtime_type"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:NCName"/>
      <xs:attribute name="version" use="required" type="xs:decimal"/>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="relation_type">
    <xs:attribute name="cardinality" use="required" type="xs:NCName"/>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
    <xs:attribute name="relationType" use="required" type="xs:NCName"/>
    <xs:attribute name="targetTypeName" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="role_type">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="role-detail" type="role-
          detail_type"/>
    </xs:sequence>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="type_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="relation" type="
          relation_type"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="event" type="
          event_type"/>
```

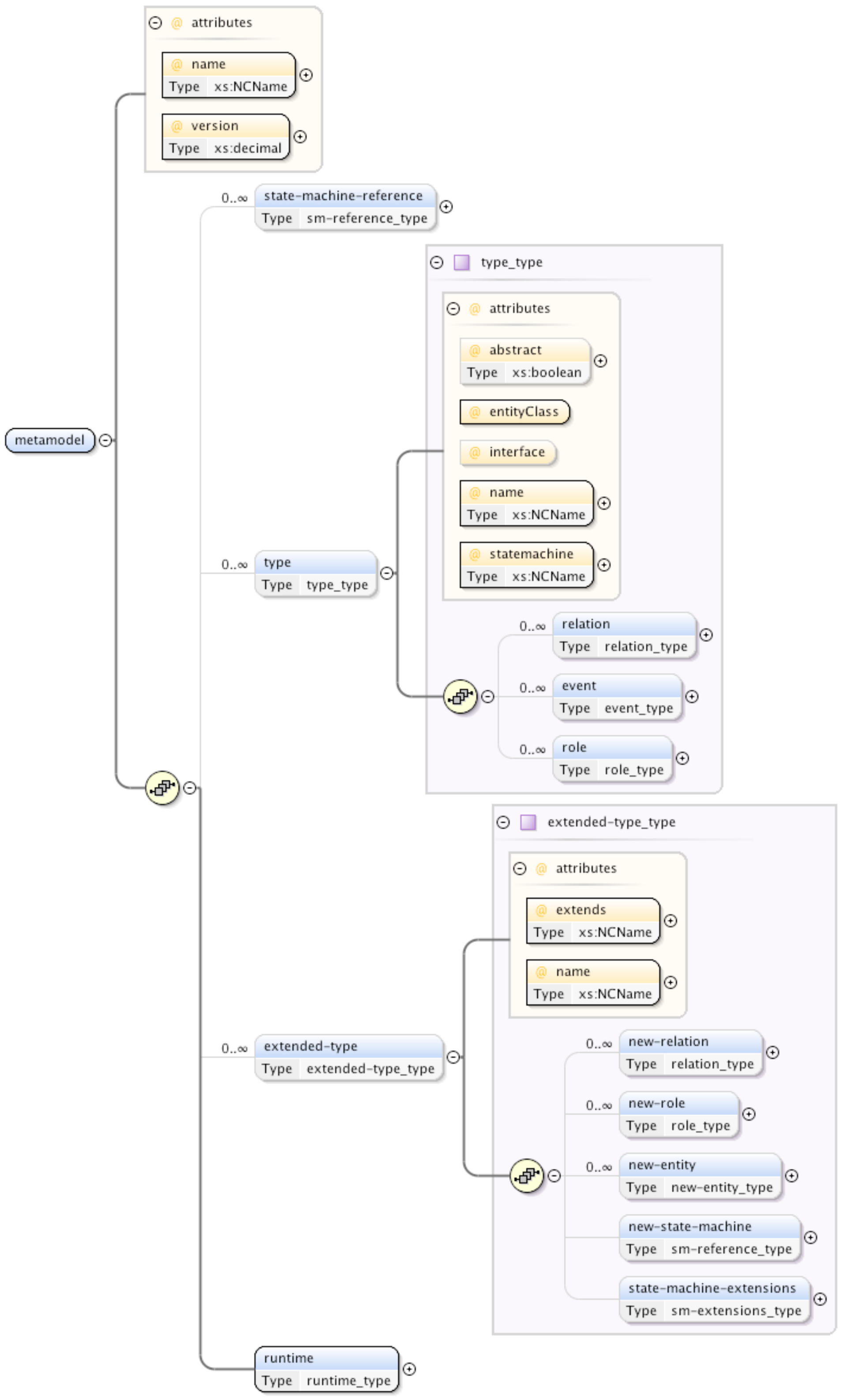


Figure B.1 Graphical representation of the metamodel definition schema

```
      <xs:element minOccurs="0" maxOccurs="unbounded" name="role" type="role_type"/>
    </xs:sequence>
    <xs:attribute name="abstract" type="xs:boolean"/>
    <xs:attribute name="entityClass" use="required"/>
    <xs:attribute name="interface"/>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
    <xs:attribute name="statemachine" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="event_type">
    <xs:attribute name="name" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="trigger-event_type">
    <xs:attribute name="event-name" use="required" type="xs:NCName"/>
    <xs:attribute name="source-name" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="runtime_type">
    <xs:attribute name="class" use="required"/>
  </xs:complexType>
  <xs:complexType name="sm-reference_type">
    <xs:attribute name="file" use="required" type="xs:NCName"/>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="new-entity_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="new-event" type="event_type"/>
    </xs:sequence>
    <xs:attribute name="entityClass" use="required"/>
    <xs:attribute name="interface" use="required"/>
  </xs:complexType>
  <xs:complexType name="sm-extensions_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="add-actions" type="add-actions_type"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="add-transition"
        type="add-transition_type"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="add-intermediate-state"
        type="add-intermediate-state_type"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="add-state" type="add-state_type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="add-actions_type">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="action" type="action_type"/>
    </xs:sequence>
    <xs:attribute name="transitionName" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="add-transition_type">
    <xs:sequence>
      <xs:element name="source-event" type="trigger-event_type"/>
      <xs:element name="before-event" type="event_type"/>
      <xs:element name="after-event" type="event_type"/>
      <xs:element minOccurs="0" maxOccurs="1" name="actions" type="actions_type"/>
    </xs:sequence>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
    <xs:attribute name="source-state" use="required" type="xs:NCName"/>
    <xs:attribute name="successor" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="add-intermediate-state_type">
    <xs:sequence>
      <xs:element name="additional-state" type="additional-state_type"/>
    </xs:sequence>
    <xs:attribute name="location" use="required" type="xs:NCName"/>
    <xs:attribute name="transitionName" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType mixed="true" name="role-detail_type">
    <xs:attribute name="type" use="required" type="xs:NCName"/>
```

```
</xs:complexType>
<xs:complexType name="basicTransition_type">
  <xs:sequence>
    <xs:element name="source-event" type="trigger-event_type"/>
    <xs:element name="before-event" type="event_type"/>
    <xs:element name="after-event" type="event_type"/>
    <xs:element minOccurs="0" name="actions" type="actions_type"/>
  </xs:sequence>
  <xs:attribute name="name" use="required" type="xs:NCName"/>
</xs:complexType>
<xs:complexType name="fullTransition_type">
  <xs:complexContent>
    <xs:extension base="basicTransition_type">
      <xs:attribute name="source-state" type="xs:NCName"/>
      <xs:attribute name="successor" type="xs:NCName"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="incomingTransition_type">
  <xs:complexContent>
    <xs:extension base="basicTransition_type">
      <xs:attribute name="source-state" type="xs:NCName"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="outgoingTransition_type">
  <xs:complexContent>
    <xs:extension base="basicTransition_type">
      <xs:attribute name="successor" type="xs:NCName"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="incoming-transitions_type">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" name="transition" type="
        incomingTransition_type"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="outgoing-transitions_type">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" name="transition" type="
        outgoingTransition_type"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="action_type">
  <xs:attribute name="class" use="required"/>
  <xs:attribute name="name" use="required" type="xs:NCName"/>
</xs:complexType>
<xs:complexType name="actions_type">
  <xs:sequence>
    <xs:element minOccurs="0" maxOccurs="unbounded" name="action" type="
        action_type"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="add-state_type">
  <xs:sequence>
    <xs:element name="incoming-transitions" type="incoming-
        transitions_type"/>
    <xs:element name="outgoing-transitions" type="outgoing-
        transitions_type"/>
  </xs:sequence>
  <xs:attribute name="enter-event" use="required" type="xs:NCName"/>
  <xs:attribute name="exit-event" use="required" type="xs:NCName"/>
  <xs:attribute name="initial-state" use="required" type="xs:boolean"/>
  <xs:attribute name="name" use="required" type="xs:NCName"/>
</xs:complexType>
<xs:complexType name="additional-state_type">
  <xs:sequence>
    <xs:element name="additional-transition" type="basicTransition_type"/>
  </xs:sequence>
  <xs:attribute name="enter-event" use="required" type="xs:NCName"/>
```

```
    <xs:attribute name="exit-event" use="required" type="xs:NCName"/>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="extended-type_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="new-relation" type="relation_type"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="new-role" type="role_type"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="new-entity" type="new-entity_type"/>
      <xs:element minOccurs="0" maxOccurs="1" name="new-state-machine" type="sm-reference_type"/>
      <xs:element minOccurs="0" maxOccurs="1" name="state-machine-extensions"
        type="sm-extensions_type"/>
    </xs:sequence>
    <xs:attribute name="extends" use="required" type="xs:NCName"/>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
  </xs:complexType>
</xs:schema>
```

B.2 State machine definition schema

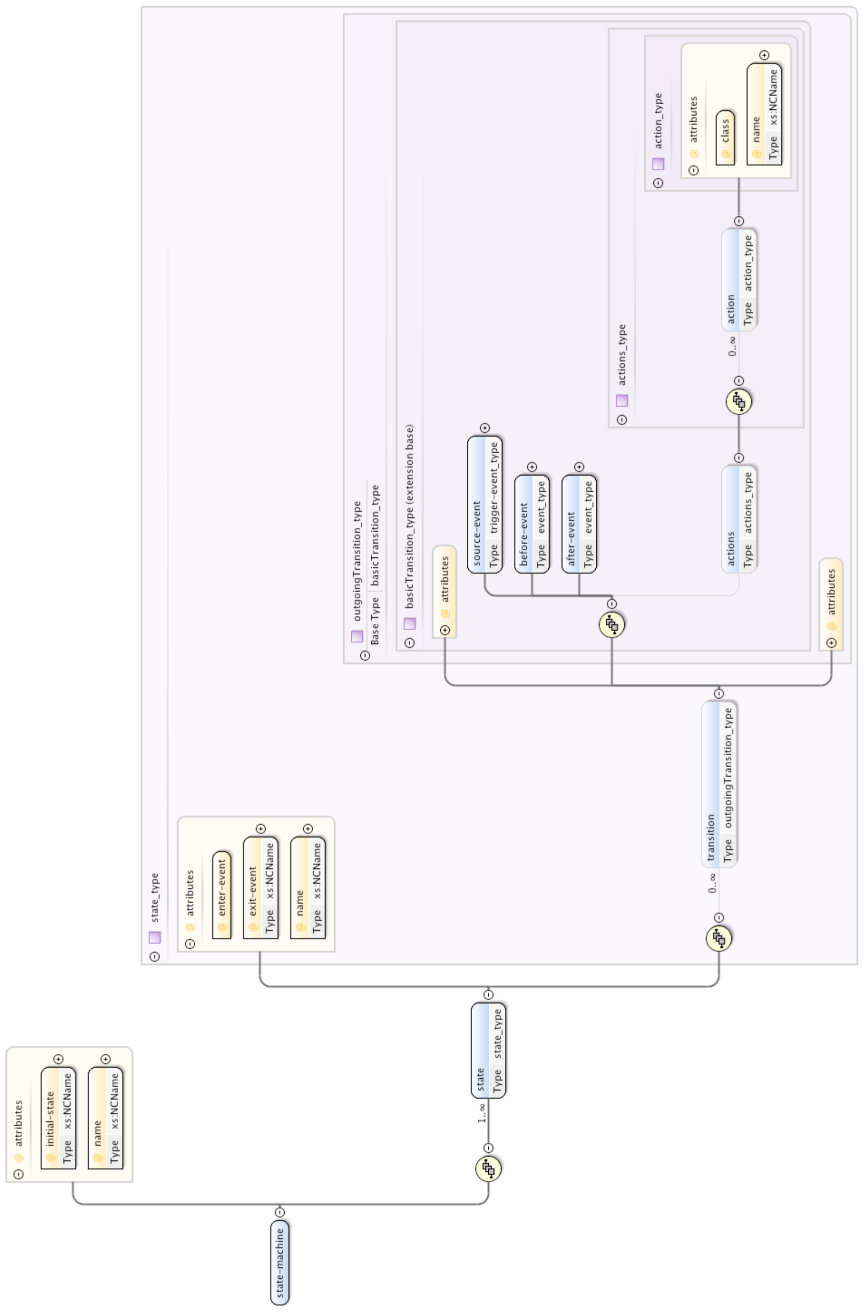

state-machine
attributes
initial-state Type xs:NCName
name Type xs:NCName
state Type state_type
1..∞
state_type
attributes
enter-event
exit-event Type xs:NCName
name Type xs:NCName
transition Type outgoingTransition_type
0..∞
outgoingTransition_type
Base Type basicTransition_type
basicTransition_type (extension base)
attributes
source-event Type trigger-event_type
before-event Type event_type
after-event Type event_type
actions Type actions_type
actions_type
action Type action_type
0..∞
action_type
attributes
class
name Type xs:NCName
attributes

Figure B.2 Graphical representation of the state machine definition schema

Listing B.2: State machine definition schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
    qualified">
    <xs:element name="state-machine">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" name="state" type="
                    state_type"/>
            </xs:sequence>
            <xs:attribute name="initial-state" use="required" type="
                xs:NCName"/>
            <xs:attribute name="name" use="required" type="xs:NCName"/>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="basicTransition_type">
        <xs:sequence>
            <xs:element name="source-event" type="trigger-event_type"/>
            <xs:element name="before-event" type="event_type"/>
            <xs:element name="after-event" type="event_type"/>
            <xs:element minOccurs="0" name="actions" type="actions_type"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:NCName" use="required"/>
    </xs:complexType>
    <xs:complexType name="outgoingTransition_type">
        <xs:complexContent>
            <xs:extension base="basicTransition_type">
                <xs:attribute name="successor" type="xs:NCName"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="event_type">
        <xs:attribute name="name" type="xs:NCName" use="required"/>
    </xs:complexType>
    <xs:complexType name="trigger-event_type">
        <xs:attribute name="event-name" type="xs:NCName" use="required"/>
        <xs:attribute name="source-name" type="xs:NCName" use="required"/>
    </xs:complexType>
    <xs:complexType name="action_type">
        <xs:attribute name="class" use="required"/>
        <xs:attribute name="name" type="xs:NCName" use="required"/>
    </xs:complexType>
    <xs:complexType name="actions_type">
        <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="action"
                type="action_type"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="state_type">
        <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="unbounded" name="transition
                "
                type="outgoingTransition_type"/>
        </xs:sequence>
        <xs:attribute name="enter-event" use="required"/>
        <xs:attribute name="exit-event" use="required" type="xs:NCName"/>
        <xs:attribute name="name" use="required" type="xs:NCName"/>
    </xs:complexType>
</xs:schema>
```

B.3 Model definition schema

Listing B.3: Model definition schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
    qualified">
  <xs:element name="definition">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="metamodel-extensions" type="metamodel-
            extensions_type"/>
        <xs:element name="runtime" type="runtime_instance_type"/>
        <xs:element name="model-structure" type="model-structure_type"/>
      </xs:sequence>
```

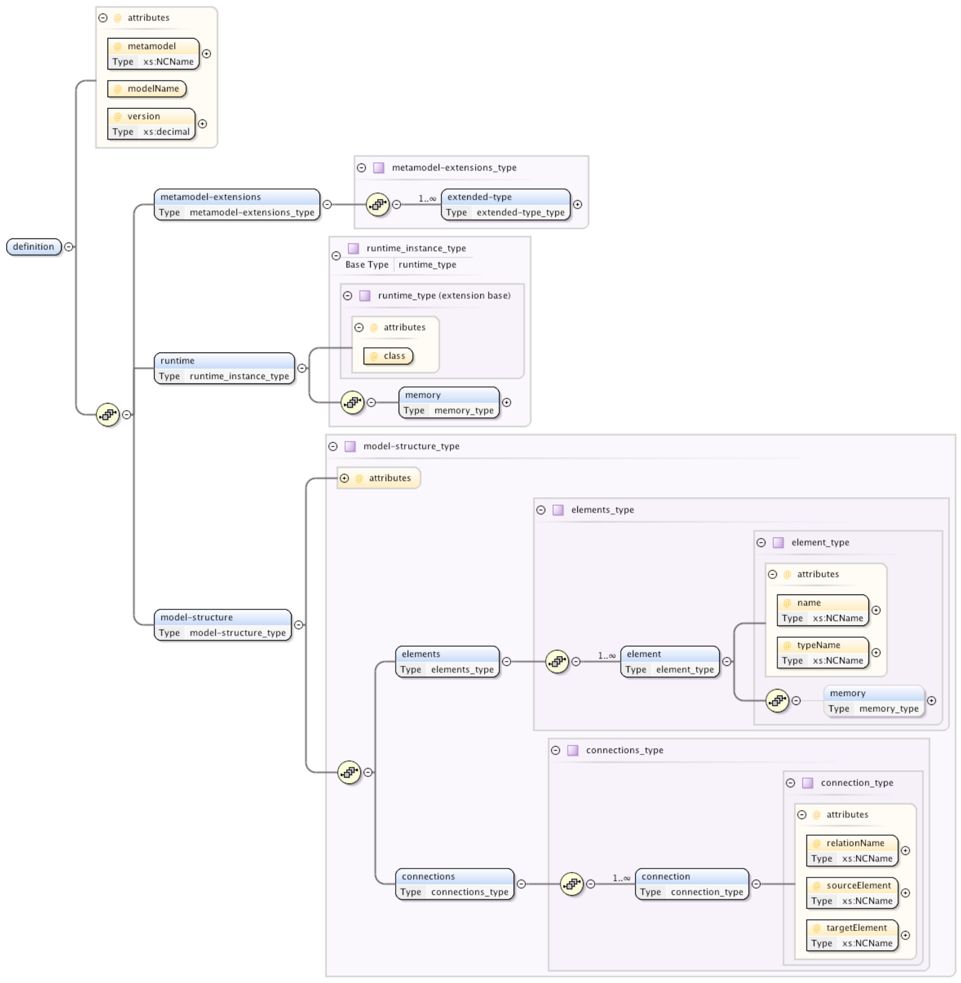


Figure B.3 Graphical representation of the model definition schema

```
      <xs:attribute name="metamodel" use="required" type="xs:NCName"/>
      <xs:attribute name="modelName" use="required"/>
      <xs:attribute name="version" use="required" type="xs:decimal"/>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="runtime_type">
    <xs:attribute name="class" use="required"/>
  </xs:complexType>
  <xs:complexType name="memory_type">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="data" type="data_type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType mixed="true" name="data_type">
    <xs:attribute name="name" use="required"/>
    <xs:attribute name="type" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="runtime_instance_type">
    <xs:complexContent>
      <xs:extension base="runtime_type">
        <xs:sequence>
          <xs:element name="memory" type="memory_type"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="model-structure_type">
```

```
    <xs:sequence>
      <xs:element name="elements" type="elements_type"/>
      <xs:element name="connections" type="connections_type"/>
    </xs:sequence>
    <xs:attribute name="root" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="elements_type">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="element" type="element_type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="element_type">
    <xs:sequence>
      <xs:element minOccurs="0" name="memory" type="memory_type"/>
    </xs:sequence>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
    <xs:attribute name="typeName" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="connections_type">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="connection" type="
          connection_type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="connection_type">
    <xs:attribute name="relationName" use="required" type="xs:NCName"/>
    <xs:attribute name="sourceElement" use="required" type="xs:NCName"/>
    <xs:attribute name="targetElement" use="required" type="xs:NCName"/>
  </xs:complexType>
  <xs:complexType name="metamodel-extensions_type">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="extended-type" type="extended-
          type_type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="extended-type_type">

   .... OMITTED ....

</xs:schema>
```

C
Navigation Language

The following specification describes the concrete syntax of the Navigation Language of Cumbia. The notation used to describe this syntax is that of JavaCC - Java Compiler Compiler [Ora10b], a parser generator based on Java. In the implementation of Cumbia we use JavaCC to describe the structure of the Navigation Language and generate a parser that builds the corresponding syntax trees. A visitor on those trees is used to interpret navigation expressions.

Table C.1 Tokens

```
<DEFAULT> SKIP : {
" "
| "\r"
| "\t"
| "\n"
}
```

```
<DEFAULT> TOKEN : {
<LT: "<" >
| <GT: ">" >
| <LE: "<=" >
| <GE: ">=" >
| <EQ: "==" >
| <IS_OF_TYPE: "isOfType" >
| <IS_ASSIGNABLE_TO_TYPE: "isAssignableToType" >
}
```

```
<DEFAULT> TOKEN : {
<ROOT: "#root" >
| <SELF: "#self" >
| <THIS: "#this" >
| <LAST: "#last" >
| <NIL: "nil" >
| <TRUE: "true" >
| <FALSE: "false" >
| <IS_TRUE: "isTrue?" >
| <IS_FALSE: "isFalse?" >
| <IS_NIL: "isNil?" >
| <IS_NOT_NIL: "isNotNil?" >
}
```

```
<DEFAULT> TOKEN : {
<IDENTIFIER: "\"" <LETTER> (<LETTER> | <DIGIT> )* "\"" >
| <RELATION: <LETTER> (<LETTER> | <DIGIT> | "_")* >
| <METHOD: <LETTER> (<LETTER> | <DIGIT> | "_")* "(" ")" >
| <NUMBER: (<DIGIT>)+>
| <#LETTER: ["$","A"-"Z","_","a"-"z","...omitted..."]>
| <#DIGIT: ["0"-"9","...omitted..."]> }
```

Table C.2 NON-Terminals

element_path	::=	local_root ("." decomposition)* <EOF>
temporal_path	::=	temporal_root ("." decomposition)*
element_value	::=	temporal_path ("::" method_call)?
local_root	::=	(<ROOT> | <SELF>)
temporal_root	::=	(<ROOT> | <THIS> | <SELF>)
method_call	::=	<METHOD>
decomposition	::=	(relation | selector)
relation	::=	<RELATION> ("[" relation_filter "]")?
relation_filter	::=	range | keys
range	::=	(<LAST> | <NUMBER>) ("-" (<LAST> | <NUMBER>))?
keys	::=	<IDENTIFIER> ("," <IDENTIFIER>)*
selector	::=	simple_selector | cond_selector | recursive_selector
cond_selector	::=	"{" conditional_expr "}"
recursive_selector	::=	"<" conditional_expr "->" temporal_path ">"
simple_selector	::=	"{?" boolean_expr "}"
conditional_expr	::=	"(" boolean_expr ")" "?" "(" temporal_path2 ")" ":" "(" temporal_path2 ")"
temporal_path2	::=	(<NIL> | temporal_path)
boolean_expr	::=	or_expr
or_expr	::=	and_expr ("||" and_expr)*
and_expr	::=	truth_expr ("&&" truth_expr)*
truth_expr	::=	("!")? boolean_value
boolean_value	::=	(boolean_operation | ("(" boolean_expr ")"))
boolean_operation	::=	(unary_operation | binary_operation)
unary_operation	::=	(op_boolean | op_relation)
op_boolean	::=	(<IS_TRUE> | <IS_FALSE>) value_boolean
value_boolean	::=	(element_value | <TRUE> | <FALSE>)
op_relation	::=	(<IS_NIL> | <IS_NOT_NIL>) temporal_path
binary_operation	::=	value_anytype (<EQ> | <LT> | <GT> | <LE> | <GE> | <IS_OF_TYPE> | <IS_ASSIGNABLE_TO_TYPE>) value_anytype
value_anytype	::=	(element_value | <IDENTIFIER> | <NUMBER>)

D
Cumbia Composition Language - CCL

The following specification describes the concrete syntax of CCL. The notation used to describe this syntax is that of Xtext [The10b], an Eclipse-based framework for the development of language infrastructures. In particular, we used Xtext to develop an editor for CCL.

Listing D.1: CCL Xtext specification

```
grammar uniandes.cumbia.ccl.CCL with org.eclipse.xtext.common.Terminals

generate CCL "http://www.cumbia.uniandes/ccl/CCL"

CCLAssembly:
  'assembly' '{'
    loadBlock=LoadBlock
    (globals+=Global)*
    (events+=OnBlock)+
  '}';

LoadBlock:
  'load' '('
    models+=ModelAliasDefinition (',' models+=ModelAliasDefinition)*
  ')' ';';

Global:
  'global' '(' variable=UntypedVariable ')' ';';

OnBlock:
  'on' ':' name=ID ('(' parameters+=TypedVariable (',' parameters+=
      TypedVariable)* ')')?
  statementBlock=StatementBlock;

StatementBlock:
  {StatementBlock}
  '{'
  (statements+=CCLStatement)*
  '}';

CCLStatement:
  FlowControl | ConpoundStatement;

ConpoundStatement:
  LineStatement ';';

LineStatement:
  VoidStatement | NonVoidStatement;

FlowControl:
  ForEach;

VoidStatement:
  VariableAssignment | ActionLink | FixReference | ResolveInInstance |
      EventLink;

NonVoidStatement:
  ModelElementOrUnknownStatement | ElementPath | SimpleTypeValue;

ModelElementOrUnknownStatement:
  NewInstance | VariableUsage | Find;

ForEach:
  name='Foreach' '(' iteratorAssignment=IteratorAssingment ')'
  statementBlock=StatementBlock;

VariableAssignment:
  variable=UntypedVariable '=' variableValue=NonVoidStatement;

IteratorAssingment:
  variable=UntypedVariable 'in' iterationPath=ElementPath;

ActionLink:
  'createActionLink' '(' name=STRING ')' '{'
    (abLinks+=ABLink)*
    (classNameActionLinks+=ClassNameActionLink)*
  '}';

ABLink:
  sourcePath=ElementPath '|' transitionName=ID '->' targetPath=ElementPath '
      ::' targetMethod=ID '(' (parameterValues+=SimpleTypeValue (','
      parameterValues+=SimpleTypeValue)*)? ')' ';';

ClassNameActionLink:
  sourcePath=ElementPath '|' transitionName=ID '->' 'createAction' '('
      actionClassName=STRING (',' argument=ElementPath)? ')' ';';

EventLink:
  'createEventLink' '(' name=STRING ')' '{'
    sourcePath=ElementPath ':' sourceEvent=ID '->' targetPath=ElementPath '|
        ' transitionName=ID ';'
  '}';

FixReference:
  name='FixReference' '(' sourceElement=ElementPath ',' targetElement=
      ElementPath ')';

ResolveInInstance:
  name='ResolveInInstance' '(' elementPath=ElementPath ',' model=
      VariableUsage ')';

SimpleTypeValue:
  StringValue | IntegerValue | BooleanValue | RealValue;

StringValue:
  value=STRING;

IntegerValue:
  value=INT;

BooleanValue:
  value=BOOLEAN;

RealValue:
  value=REAL;

NewInstance:
  'new' model=[ModelAliasDefinition] ('{' memoryData=MemoryData '}')?;
```

```

MemoryData:
  {MemoryData}
  (data+=MemoryParam)*;

MemoryParam:
  param=ID '=' value=STRING ';';

VariableUsage:
  variableReference=[VariableDeclaration];

Find:
  'findByName' '(' model=VariableUsage ',' elementName=STRING ')';

ElementPath:
  elementRoot=ModelElementOrUnknownStatement ('[' expression=STRING ']')?;

VariableDeclaration:
  TypedVariable | UntypedVariable;

TypedVariable:
  ModelVariable | ElementVariable | GenericVariable | ModelAliasVariable;

UntypedVariable:
  name=ID;

ModelVariable:
  type=[ModelAliasDefinition] name=ID;

ModelAliasVariable:
  alias=ID name=ID;

ElementVariable:
  type=TypeElement name=ID;

GenericVariable:
  type=TypeSimple name=ID;

ModelAliasDefinition:
  domainName=ID ':' modelName=ID name=ID;

TypeElement:
  domainName=ID '::' elementName=ID;

TypeSimple:
  domainName=TypeSimpleEnum;

enum TypeSimpleEnum:
  string='String' | integer='Integer' | real='Real' | boolean='Boolean';

BOOLEAN:
  'True' | 'False';

REAL:
  INT '.' INT;
```

Bibliography

[AAD$^+$07] Ashish Agrawal, Mike Amend, Manoj Das, Mark Ford, Chris Keller, Matthias Kloppmann, Dieter König, Frank Leymann, Ralf Müller, Gerhard Pfau, Karsten Plösser, Ravi Rangaswamy, Alan Rickayzen, Michael Rowley, Patrick Schmidt, Ivana Trickovic, Alex Yiu, and Matthias Zeller. WS-BPEL Extension for People (BPEL4People), Version 1.0, June 2007. `http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/`, 2007.

[ABB$^+$05] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie B. Carson, Ian Evans, Dale Green, Kim Haase, and Eric Jendrock. Expression language. In *The J2EE 1.4 Tutorial*, chapter 12: JavaServer Pages Technology, pages 499–506. Sun Microsystems, Inc., Santa Clara, California, USA, December 2005.

[Abo06] Faisal Abouzaid. A Mapping from Pi-Calculus into BPEL. In *Proceeding of the 2006 conference on Leading the Web in Concurrent Engineering*, pages 235–242, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.

[AD97] Larry Apfelbaum and John Doyle. Model Based Testing. Software Quality Week Conference, May 1997.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[AHS93] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5:23–70, February 1993.

[AHST97a] Gustavo Alonso, Claus Hagen, Hans-Jörg Schek, and Markus Tresch. Distributed processing over stand-alone systems and applications. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 575–579, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[AHST97b] Gustavo Alonso, Claus Hagen, Hans-Jörg Schek, and Markus Tresch. Towards a Platform for Distributed Application Development. In *NATO Advance Studies Institute (ASI)*, pages 195–221. Springer, 1997.

[AWB$^+$93] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In *European Conference on Object Oriented*

Programming, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer-Verlag, 1993.

[Bar02] Patrick Barril. Net Execution. In Claude Girault and Rüdifer Valk, editors, *Petri Nets for Systems Engineering*, chapter 20, pages 417–431. Springer Verlag, 2002.

[Bar07] Graham Barber. Open SOA, Service Component Architecture Home. `http://www.osoa.org/display/Main/Service+Component+Architecture+Home`, November 2007.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BCF$^+$07] Scott Boag, Don Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery/`, January 2007.

[BCF$^+$08] Christopher Brooks, Chihhong Cheng, Thomas H. Feng, Edward A. Lee, and Reinhard von Hanxleden. Model engineering using multimodeling. In *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM '08)*, October 2008.

[BCP07] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. Behavioural Types for Service Integration: Achievements and Challenges. *Electronic Notes in Theoretical Computer Science*, 180:41–54, June 2007.

[BFG95] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Practical Use of Graph Rewriting. Technical Report 95-373, Queen's University, Department of Computing and Information Science, January 1995.

[Biz10] BizAgi. Introduction to BizAgi. `http://wiki.bizagi.com/en/index.php?title=Introduction`, 2010.

[BV07] Tom Baeyens and Miguel Valdes. The Process Virtual Machine. `http://docs.jboss.com/jbpm/pvm/article/`, 2007.

[BVJ$^+$06] Mathieu Braem, Kris Verlaenen, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, Eddy Truyen, Wouter Joosen, and Viviane Jonckers. Isolating Process-Level Concerns Using Padus. In Schahram Dustdar, José Fiadeiro, and Amit Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 113–128. Springer Berlin / Heidelberg, 2006.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath), version 1.0. `http://www.w3.org/TR/xpath/`, November 1999.

[CKO92] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[Cle07a] Thomas Cleenewerck. Implementing Languages with Plans for Growth. In *The 6th Belgian-Netherlands Software Evolution Workshop - Benevol 2007*. University of Namur, Belgium, 2007.

[Cle07b] Thomas Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universiteit Brussel, 2007.

[CM06] Anis Charfi and Mira Mezini. Aspect-Oriented Workflow Languages. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *Lecture Notes in Computer Science*, pages 183–200. Springer Berlin / Heidelberg, 2006.

[Coa99] Workflow Management Coalition. WFMC-TC-1011 Ver 3 Terminology and Glossary English. `http://www.wfmc.org`, February 1999.

[Coa08] Workflow Management Coalition. Process Definition Interface – XML Process Definition Language, Version 2.1. `http://www.wfmc.org`, March 2008.

[Coa09] Workflow Management Coalition. XPDL Implementations. `http://www.wfmc.org/xpdl-implementations.html`, 2009.

[Cor07] Darío Correal. *Definition and execution of multiple viewpoints on workflow processes*. PhD thesis, Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia, 2007.

[Cra07] Iain D. Craig. *Object-Oriented Programming Languages: Interpretation*, volume VIII of *Undergraduate Topics in Computer Science*. Springer-Verlag, London, 2007.

[Cro06] D. Crockford. The application/Json Media Type for JavaScript Object Notation (JSON), Request for Comments: 4627. `http://www.ietf.org/rfc/rfc4627.txt`, July 2006.

[CV08] Nadya Calderón and Carlos Vega. Composición y adaptación de modelos ejecutables extensibles para aplicaciones eLearning : caso IMS-LD. Master's thesis, Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia, 2008.

[CvdBE05] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Modeling Aspect-Oriented Compositions. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 100–109. Springer Berlin / Heidelberg, October 2005.

[CvdBE07] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. International Conference on AspectOriented Software Development, Industry Track, October 2007.

[Dav04] Drew Davidson. OGNL Language Guide. `http://www.opensymphony.com/ognl/html/LanguageGuide/index.html`, 2004.

[DEA98] Samir Dami, Jacky Estublier, and Mahfound Amiour. APEL: A Graphical Yet Executable Formalism for Process Modeling. *Automated Software Engineering*, 5:61–96, January 1998.

[Dee07] Ewa Deelman. *Looking into the Future of Workflows: The Challenges Ahead*, chapter 28, pages 475–481. Springer, London, 2007.

[Der06] Dirk Deridder. *A Concept-Centric Environment for Software Evolution in an Agile Context.* PhD thesis, Vrije Universiteit Brussel, Brussel, 2006.

[DM05] Yi Duan; and Huadong Ma. Modeling flexible workflow based on temporal logic. In *Computer Supported Cooperative Work in Design*, volume 1, pages 508–513, 2005.

[DRS94] Roger Duke, Gordon Rose, and Graeme Smith. Object-z: a specification language advocated for the description of standards. Technical Report 94-95, Department of Computer Science, The University of Queensland, December 1994.

[EIV05] Jacky Estublier, Anca Daniela Ionita, and Germán Vega. Composing Domain-Specific Languages for Wide-scope Software Engineering Applications. In *Proceedings of MoDELS 2005*, Lecture Notes in Computer Science, pages 69–83. Springer, October 2005.

[EIV06] Jacky Estublier, Anca Daniela Ionita, and Germán Vega. Relationships for Domain Reuse and Composition. *Journal of Research and Practice in Information Technology*, 38:287–302, 2006.

[EN96] Clarence A. Ellis and Gary J. Nutt. Workflow: The Process Spectrum. In Amit Sheth, editor, *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems*, pages 140–145, May 1996.

[Esp09] John Espitia. Una herramienta basada en Cumbia para análisis dinámico de modelos SCA. Master's thesis, Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia, 2009.

[Eva03] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Addison-Wesley, Boston, MA, USA, 2003.

[EVLA$^+$03] Jacky Estublier, Jorge Villalobos, Tuyet Le Anh, Sonia Jamal-Sanlaville, and German Vega. An Approach and Framework for Extensible Process Support System. In Flávio Oquendo, editor, *9th International Workshop on Software Process Technology 2003*, volume 2786 of *Lecture Notes in Computer Science*, pages 46–61. Springer, September 2003.

[EW01] Rik Eshuis and Roel Wieringa. A comparison of Petri net and activity diagram variants. Technical report, DFG Research Group "Petri Net Technology", September 2001.

[FBJ$^+$05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: a generic model weaver. In *Proceedings of the 1ere Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, 2005.

[FCS04] Sérgio M. Fernandes, Joao Cachopo, and António R. Silva. Supporting Evolution in Workflow Definition Languages. In *SOFSEM 2004: Theory and Practice of Computer Science*, volume 2932 of *Lecture Notes in Computer Science*, pages 49–59. Springer, Berlin / Heidelberg, February 2004.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.

[FF04] Diogo M. R. Ferreira and J. J. Pinto Ferreira. Developing a reusable workflow engine. *Journal of Systems Architecture: the EUROMICRO Journal*, 50(6):309–324, June 2004.

[FLR08] Fabricio Fernandes, Frédéric Lepage, and Jean-Claude Royer. The STSLib Tutorial. `http://www.emn.fr/x-info/jroyer/WEBLIB/index.html`, September 2008.

[FR07] France, Robert and Rumpe, Bernhard. Model-driven Development of Complex Software: A Research Roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA, May 2007. IEEE Computer Society.

[FR08] Fabricio Fernandes and Jean-Claude Royer. The STSLib Project: Towards a Formal Component Model Based on STS. *Electronic Notes in Theoretical Computer Science*, 215:131–149, June 2008.

[Fra02] Frank, Ulrich. Multi-perspective Enterprise Modeling (MEMO) - Conceptual Framework and Modeling Languages. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, volume 3, pages 1258–1267, Washington, DC, USA, 2002. IEEE Computer Society.

[GdV98] Paul Grefen and Remmert R. de Vries. A Reference Architecture for Workflow Management Systems. *Data & Knowledge Engineering*, 27(1):31–57, 1998.

[GHS95] Dimitrios Georgakopoulos, Mark F. Hornick, and Amit P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[GKD01] A. Goh, Y. K. Koh, and D. S. Domazet. ECA rule-based support for workflows. *Artificial Intelligence in Engineering*, 15(1):37 – 46, January 2001.

[GKP98] Robert Geisler, Marcus Klar, and Claudia Pons. Dimensions and Dichotomy in Metamodeling. Technical Report 98-5, Technical University Berlin, 1998.

[GP07] Steve Gregory and Martha Paschali. A prolog-based language for workflow programming. In Amy Murphy and Jan Vitek, editors, *COORDINATION'07: Proceedings of the 9th international conference on Coordination models and languages*, volume 4467, pages 56–75, Berlin, Heidelberg, 2007. Springer-Verlag.

[GT08] Denis Gagné and André Trudel. A temporal semantics for workflow control patterns. In *COMPSAC '08: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 999–1004, Washington, DC, USA, 2008. IEEE Computer Society.

[GV01] Claude Girault and Rudiger Valk. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[GZ05] Leif Geiger and Albert Zündorf. Statechart Modeling with Fujaba. *Electronic Notes in Theoretical Computer Science*, 127(1):37–49, 2005.

[HBJ09] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. COPE - Automating Coupled Evolution of Metamodels and Models. In Sophia Drossopoulou, editor, *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *Lecture Notes in Computer Science*, pages 52–76. Springer, 2009.

[HBMN07] Cecile Hardebolle, Frederic Boulanger, Dominique Marcadet, and Guy V. Naquet. A generic execution framework for models of computation. In *MOMPES '07: Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 45–54, Washington, DC, USA, 2007. IEEE Computer Society.

[Hol95] David Hollingsworth. The Workflow Reference Model - TC00-1003, 1995.

[Hol04] David Hollingsworth. The Workflow Reference Model: 10 Years On. In Layna Fischer, editor, *Workflow Handbook 2004*, pages 295–312. Future Strategies Inc., 2004.

[HSS05] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri Nets. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Proceedings of the 3rd International Conference on Business Process Management (BPM 2005)*, pages 220–235, Nancy, France, 2005. Springer Verlag.

[HVD08] Ta'id Holmes, Martin Vasko, and Schahram Dustdar. VieBOP: Extending BPEL Engines with BPEL4People. In *16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 547–555. IEEE Computer Society, February 2008.

[HWS$^+$06] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. *Nucleid Acids Research*, 34(Web server issue):729–732, July 2006.

[IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology. Technical Report Std 610.12-1990, Institute of Electrical and Electronics Engineers, 1990.

[IMS01] IMS Global Learning Consortium. IMS Learner Information Packaging Information Model Specification, version 1.0. `http://www.imsglobal.org/profiles/lipinfo01.html`, March 2001.

[IMS02] IMS Global Learning Consortium. IMS Reusable Definition of Competency or Educational Objective - Information Model, version 1.0. `http://www.imsglobal.org/competencies/rdceov1p0/imsrdceo_infov1p0.html`, October 2002.

[IMS03a] IMS Global Learning Consortium. IMS-CP Content Packaging Information Model, version 1.0. `http://www.imsglobal.org/content/packaging`, 2003.

[IMS03b] IMS Global Learning Consortium. IMS Learning Design, version 1.0. `http://www.imsglobal.org/learningdesign/`, February 2003.

[IMS05] IMS Global Learning Consortium. IMS ePortfolio Information Model, version 1.0. `http://www.imsglobal.org/ep/epv1p0/imsep_infov1p0.html`, June 2005.

[Ins08] Institute for Software Integrated Systems. GME: Generic Modeling Environment. `http://www.isis.vanderbilt.edu/projects/gme/`, 2008.

[Jab94] Stefan Jablonski. MOBILE: A Modular Workflow Model and Architecture. In *Proceedings of the Fourth International Working Conference on Dynamic Modelling and Information Systems*, pages 1–30, Noordwijkerhout, Netherlands, 1994.

[JFB08] Cédric Jeanneret, Robert France, and Benoit Baudry. A reference process for model composition. In *AOM '08: Proceedings of the 2008 AOSD Workshop on Aspect-Oriented Modeling*, pages 1–6, New York, NY, USA, 2008. ACM.

[Jim07] Camilo Jiménez. Composición de modelos ejecutables extensibles en una fábrica de aplicaciones basadas en workflows. Master's thesis, Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia, 2007.

[jso10] json.org. Introducing JSON. `http://www.json.org/`, 2010.

[JVR97] Rushikesh K. Joshi, N. Vivekananda, and D. Janaki Ram. Message filters for object-oriented systems. *Software – Practice and Experience*, 27(6):677–699, June 1997.

[KAR06] Ekkart Kindler, Björn Axenath, and Vladimir Rubin. AMFIBIA: A Meta-Model for the Integration of Business Process Modelling Aspects. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil M. P. van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, volume 06291 of *Dagstuhl Seminar Proceedings.* Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), S chloss Dagstuhl, Germany, 2006.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.

[KGK+07] Dierk Koening, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*, chapter 7: Using power features, pages 208–212. Manning Publications, 2007.

[Kie03] Bartosz Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows.* PhD thesis, Centre for Cooperative Information Systems, Queensland University of Technology, Brisbane, Australia, February 2003.

[KK02] Karagiannis, Dimitris and Harald Kühn. Metamodelling Platforms. In Bauknecht, K., Min Tjoa, A., and Quirchmayer, G., editors, *EC-WEB '02: Proceedings of the Third International Conference on E-Commerce and Web Technologies*, volume 2455 of *Lecture Notes in Computer Science*, pages 451–464, Berlin, Heidelberg, 2002. Springer-Verlag.

[KKM09] Matthias Kloppmann, Dieter König, and Simon Moser. The Dichotomy of Modeling and Execution: BPMN and WS-BPEL. In Wil M. P. van der Aalst and Jorge Cardoso, editors, *Handbook of Research on Business Process Modeling*, chapter 4, pages 70–91. Information Science Reference, April 2009.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model transformation in the large. In *ESEC-FSE '07: Proceedings of the the*

6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 285–294, New York, NY, USA, 2007. ACM.

[KMF90] J. Kramer, J. Magee, and A. Finkelstein. A constructive approach to the design of distributed systems. In *10th International Conference on Distributed Computing Systems*, pages 580 –587. IEEE, 1990.

[KMSF01] David Kortenkamp, Tod Milam, Reid Simmons, and Joaquin Fernandez. Collecting and analyzing data from distributed control programs. In *Proceedings of the Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 133–151. Elsevier Publishing, 2001.

[LBM$^+$01] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Nordstrom, Greg, Sprinkle, Jonathan, and Gábor Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, November 2001.

[LHS03] Brad Long, Daniel Hoffman, and Paul Stropper. Tool Support for Testing Concurrent Java Components. *IEEE Transactions on Software Engineering*, 29(6):555–566, June 2003.

[LKT04] Janne Luoma, Steven Kelly, and Juha P. Tolvanen. Defining Domain-Specific Modeling Languages: Collected Experiences. Technical Report TR-33, University of Jyväskylä, Finland, October 2004.

[LM07] Roberto Lucchi and Manuel Mazzara. A PI-Calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96 – 118, 2007. Web Services and Formal Methods.

[Man01] Dragos-Anton Manolescu. *Micro-Workflow: a workflow architecture supporting compositional object-oriented software development.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2001.

[Már08] Pablo Márquez. Composición de modelos ejecutables para la construcción de motores de lenguajes específicos. Master's thesis, Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia, 2008.

[MB02] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture.* Addison-Wesley Object Technology Series. Addison-Wesley Professional, May 2002.

[MBZL09] Timothy McPhillips, Shawn Bowers, Daniel Zinn, and Bertram Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541–551, May 2009.

[MD89] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. *SIGMOD Rec.*, 18(2):215–224, 1989.

[Mic07] Microsoft Corporation. Windows Workflow Foundation - .NET Framework 3.5. `http://msdn.microsoft.com/en-us/library/ms735967.aspx`, 2007.

[MnCVC07] Olga Mariño, Rubby Casallas, Jorge Villalobos, and Darío Correal. Bridging the Gap between e-learning Modeling and Delivery through the Transformation of Learnflows into Workflows. In Samuel Pierre, editor, *E-learning networked. Environments and Architectures: A Knowledge Processing Perspective*, Advanced Information and Knowledge Processing, chapter 2, pages 27–59. Springer-Verlag New York, LLC, 2007.

[Muñ09] Manuel Muñoz. Motor de aspectos que resuelve interferencias basado en Cumbia : caso BPEL. Master's thesis, Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia, 2009.

[MWGW99] Peter Muth, Jeanine Weißenfels, Michael Gillmann, and Gerhard Weikum. Workflow history management in virtual enterprises using a light-weight workflow management system. In *Proceedings of the Ninth International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises (RIDE 99)*, volume -, pages 148–155, Sydney, Australia, March 1999. IEEE Press.

[Nut96] Gary J. Nutt. The evolution towards flexible workflow systems. *Distributed System Engineering*, 3:276–294, 1996.

[OAS05] OASIS Technical Committee. Web Services Business Process Execution Language, Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`, 2005.

[Obj05] Object Management Group. Software Process Engineering Metamodel Specification, version 1.0. `http://www.omg.org/cgi-bin/doc?formal/02-11-14`, January 2005.

[Obj06a] Object Management Group. Meta Object Facility (MOF) Core Specification, V2.0. `http://www.omg.org/spec/MOF/2.0/`, 2006.

[Obj06b] Object Management Group. Object Constraint Language, version 2.0. `http://www.omg.org`, May 2006.

[Obj07a] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/`, November 2007.

[Obj07b] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. `http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/`, November 2007.

[Obj08] Object Management Group. Business Process Modeling Notation, V1.1. http://www.omg.org/spec/BPMN/1.1/PDF, 2008.

[Obj09a] Object Management Group. Business Process Model and Notation (BPMN), Beta 1 for Version 2.0. http://www.omg.org/spec/BPMN/2.0, August 2009.

[Obj09b] Object Management Group. List of BPMN Supporters. http://www.omg.org/bpmn/BPMN_Supporters.htm, 2009.

[Obj10] Object Management Group. MDA Specifications. http://www.omg.org/mda/specs.htm, 2010.

[OGA$^+$06] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation : Practice and Experience*, 18(10):1067–1100, August 2006.

[Ora09] Oracle. Oracle, Business Activity Monitoring. http://www.oracle.com/appserver/business-activity-monitoring.html, 2009.

[Ora10a] Oracle. JAR File Specification. http://download.oracle.com/javase/1.5.0/docs/guide/jar/jar.html, 2010.

[Ora10b] Oracle. Java Compiler Compiler - The Java Parser Generator. https://javacc.dev.java.net/, 2010.

[OSO07] OSOA Collaboration. SCA Service Component Architecture - Assembly Model Specification, Version 1.00. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications, March 2007.

[Öve98] Gunnar Övergaard. A formal approach to relationships in the Unified Modeling Language. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Software Modelling Techniques*, number 19803 in TUM, Germany, April 1998. Technische Universität München.

[OW208] OW2 Consortium. Orchestra: Open Source BPEL / BPM Solution. http://orchestra.ow2.org/xwiki/bin/view/Main/WebHome, 2008.

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329 – 400, 1998.

[Pau04] Cesare Pautasso. *A Flexible System for Visual Service Composition.* PhD thesis, Swiss Federal Institute of Technology Zurich, Zurich, Switzerland, 2004.

[Pau09] Cesare Pautasso. Compiling Business Process Models into Executable Code. In Jorge Cardoso and Wil M. P. van der Aalst, editors, *Handbook of Research in Business Process Management*, chapter 15, pages 318–337. IGI Global, 2009.

[PE08] Gabriel Pedraza and Jacky Estublier. An Extensible Services Orchestration Framework through Concern Composition. International Workshop on Non-functional System Properties in Domain Specific Modeling Languages, September 2008.

[Ped09] Gilberto Pedraza. Evaluación de la Extensibilidad de Modelos Ejecutables - Caso Portafolio en Cumbia LD. Master's thesis, Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia, 2009.

[Pér09] Javier Igua Pérez. Properties of Building an Extensible Workflow Engine: An exploration in open systems. Master's thesis, Vrije Universiteit Brussel, Brussel, August 2009.

[PPL01] Martin Purvis, Maryam Purvis, and Selena Lemalu. A Framework for Distributed Workflow Systems. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, volume 9, page 9039, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[PW05] Frank Puhlmann and Mathias Weske. Using the Pi-Calculus for Formalizing Workflow Patterns. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Proceedings of the 3rd International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 153–168. Springer-Verlag, 2005.

[RA81] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[Ram06] Norman Ramsey. ML Module Mania: A Type-Safe, Separately Compiled, Extensible Interpreter. *Electronic Notes in Theoretical Computer Science*, 148:181–209, March 2006.

[Ric10] Mathias Ricken. Concjunit. `http://www.cs.rice.edu/~mgricken/research/concutest/concjunit/`, 2010.

[Rom07] Daniel Romero. Modelos ejecutables extensibles como activos en una fábrica de motores de workflow: caso BPEL. Master's thesis, Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia, 2007.

[RSV11] Carlos Rodríguez, Mario Sánchez, and Jorge Villalobos. Executable model composition - A multilevel approach. To be published in the proceedings of SAC'11: The 2011 ACM Symposium on Applied Computing, March 2011.

[RtHEvdA04] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow Data Patterns. Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[RtHEvdA05] Nick Russell, Arthur ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In Lois Delcambre, Christian Kop, Heinrich Mayr, John Mylopoulos, and Oscar Pastor, editors, *Conceptual Modeling – ER 2005*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368. Springer Berlin / Heidelberg, 2005.

[RtHEvdA07] Nick Russel, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. newYAWL: achieving comprehensive patterns support in workflow for the controlflow data an resource perspectives. Technical Report BPM-07-05, BPM Center, 2007.

[RtHvdAM06] Nick Russell, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and Natalya Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM-06-22, BPMcenter.org, 2006.

[RvdAtH06] Nick Russell, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. Exception Handling Patterns in Process-Aware Information Systems. Technical Report BPM-06-04, BPMcenter.org, 2006.

[RvtE05] Russell, Nick, van der Aalst, Wil M. P., ter Hofstede, Arthur H. M., and Edmond, David. Workflow Resource Patterns: Identification, Representation and Tool Support. In Oscar Pastor and Joao Falcao, editors, *Lecture Notes in Computer Science*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232–232, Heidelberg, 2005. Springer Berlin.

[Sch71] Stephen A. Schuman. An extensible interpreter. In *Proceedings of the international symposium on Extensible languages*, pages 120–128, New York, NY, USA, 1971. ACM.

[SJV10] Mario Sánchez, Camilo Jiménez, and Jorge Villalobos. Model Based Testing for Workflow Enabled Applications. To be published in: Revista Computación y Sistemas, México, 2010.

[SJVD09] Mario Sánchez, Camilo Jiménez, Jorge Villalobos, and Dirk Deridder. Extensibility in Model-based Business Process Engines. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS EUROPE 2009*, volume 33 of *LNBIP*, pages 157–174, Berlin - Heidelberg, July 2009. Springer-Verlag.

[SVR09] Mario Sánchez, Jorge Villalobos, and Daniel Romero. A State Machine Based Coordination Model applied to Workflow Applications. *Avances en Sistemas e Informática*, 6(1):35–44, June 2009.

[The06] The Codehaus. Language Guide for MVEL 2.0. http://mvel.codehaus.org/Language+Guide+for+2.0, 2006.

[The08] The YAWL Foundation. YAWL - User Manual, v2.0. http://yawlfoundation.org/, 2008.

[The10a] The Eclipse Foundation. Eclipse Modeling Framework Project. http://www.eclipse.org/modeling/emf/, 2010.

[The10b] The Eclipse Foundation. Xtext - Language Development Framework. http://www.eclipse.org/Xtext/, 2010.

[UPL06] Mark Utting, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-Based Testing. Technical Report 04/2006, University of Waikato, Department of Computer Science, Hamilton, New Zealand, April 2006.

[vdA99] Wil M. P. van der Aalst. Woflan: a Petri-net-based workflow analyzer. *Syst. Anal. Model. Simul.*, 35(3):345–357, 1999.

[vdADK02] Wil M. P. van der Aalst, Jörg Desel, and Ekkart Kindler. On the semantics of EPCs: A vicious circle. In M. Nüttgens and F. J. Rump, editors, *Proceedings of the EPK 2002: Business Process Management using EPCs*, pages 71–79, 2002.

[vdAtH06] Wil M.P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: Yet Another Workflow Language (Revised Version). Technical Report FIT-TR-2003-04, Queensland University of Technology, Brisbane, 2006.

[vdAtHKB03] Wil M.P. van der Aalst, Arthur H. M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Technical Report BPM-03-06, BPMcenter.org, 2003.

[vdAtHW03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. Business Process Management: a survey. In Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske, editors, *Proceedings of the 2003 International Conference on Business Process Management (BPM2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12, Berlin, 2003. Springer-Verlag.

[vdAvH02] Wil M. P. van der Aalst and Kees van Hee. *Workflow Management - Models, Methods, and Systems.* Cooperative Information Systems. MIT Press, 2002.

[vDJVVvdA07] B. F. van Dongen, M. H. Jansen-Vullers, H. M. W. Verbeek, and Wil M. P. van der Aalst. Verification of the SAP reference models using EPC reduction, state-space analysis, and invariants. *Comput. Ind.*, 58(6):578–601, 2007.

[vDK98] Arie van Deursen and Paul Klint. Little languages: little maintenance. *Journal of Software Maintenance*, 10(2):75–92, 1998.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[VvdA00] E. Verbeek and Wil M. P. van der Aalst. Woflan 2.0: A Petri-net-based workflow diagnosis tool. In M. Nielsen and D. Simpson, editors, *21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), Aarhus, Denmark*, volume 1825 of *Lecture Notes in Computer Science*, pages 475–484. Springer-Verlag, Berlin, 2000.

[WEB$^+$07] Bruno Wassermann, Wolfgang Emmerich, Ben Butchart, Nick Cameron, Liang Chen, and Jignesh Patel. *Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling*, chapter 26, pages 428–449. Springer, London, 2007.

[Wes07] Mathias Weske. *Business Process Management - Concepts, Languages, Architectures.* Springer, November 2007.

[WEvdAtH05] Moe Thandar Wynn, David Edmond, Wil M. P. van der Aalst, and Arthur H.M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets. In Gianfranco Ciardo and Philippe Darondeau, editors, *Proceedings of the 26th International Conference On Application and Theory of Petri Nets and Other Models of Concurrency*, volume 3536 of *Lecture Notes in Computer Science*, pages 423–443. Springer, 2005.

[WG07] Peter Y. H. Wong and Jeremy Gibbons. A process-algebraic approach to workflow specification and refinement. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2007.

[WLC$^+$05] Yi Wang, Minglu Li, Jian Cao, Feilong Tang, Lin Chen, and Lei Cao. An ECA-Rule-Based Workflow Management Approach for Web Services Composition. In Hai Zhuge and Geoffrey Fox, editors, *Grid and Cooperative Computing - GCC 2005*, volume 3795 of *Lecture Notes in Computer Science*, pages 143–148. Springer, 2005.

[Wora] Workflow Patterns Initiative. Pattern 9 (Structured Discriminator). `http://www.workflowpatterns.com/patterns/control/advanced_branching/wcp9.php`.

[Worb] Workflow Patterns Initiative. Workflow Patters, Commercial Products Evaluations. `http://www.workflowpatterns.com/evaluations/commercial/index.php`.

[Worc] Workflow Patterns Initiative. Workflow Patters, Open Source Products Evaluations. `http://www.workflowpatterns.com/evaluations/opensource/index.php`.

[Word] Workflow Patterns Initiative. Workflow Patters, Standard Evaluations. `http://www.workflowpatterns.com/evaluations/standard/index.php`.

[wsp07] wsper.org. WS-BPEL 2.0 metamodel. `http://www.ebpml.org/wsper/wsper/ws-bpel20.html`, 2007.

[WvdAD+06] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. Pattern-based Analysis of BPMN - An extensive evaluation of the Control-flow, the Data and the Resource Perspectives. Technical Report BPM-05-26, BPM Center, 2006.

[WvdADH03] P. Wohed, Wil M. P. van der Aalst, M. Dumas, and Arthur H. M. Ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In I. Y. Song, S. W. Liddle, T. W. Ling, and P. Scheuermann, editors, *Conceptual Modeling - ER 2003*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2003.

[WWWKD96] Jeanine Weißenfels, Dirk Wodtke, Gerhard Weikum, and Angelika Kotz Dittrich. The MENTOR Architecture for Enterprise-wide Workflow Management. In A. Sheth, editor, *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems*, May 1996.

[zM04] Michael zur Muehlen. *Workflow-based Process Controlling*. Advances in Information Systems and Management Science. Logos Verlag, Berlin, 2004.

[zMR08] Michael zur Muehlen and Jan Recker. How Much Language is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. In Zohra Bellahsène and Michel Léonard, editors, *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE 2008)*, volume 5074 of *Lecture Notes in Computer Science*, pages 465–479, Berlin, Heidelberg, June 2008. Springer-Verlag.