A Categorical Framework for the Specification and the Verification of Aspect Oriented Systems

par
Arsène Sabas

Département d’informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences
en vue de l’obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Juillet, 2012

© Arsène Sabas, 2012.
Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

A Categorical Framework for the Specification and the Verification of Aspect Oriented Systems

présentée par:

Arsène Sabas

a été évaluée par un jury composé des personnes suivantes:

Houari Sahraoui, président-rapporteur
Michel Boyer, directeur de recherche
Michael Barr, codirecteur
Geña Hahn, membre du jury
Marc Aiguier, examinateur externe
Houari Sahraoui, représentant du doyen de la F.A.S.

Thèse acceptée le: 2 Octobre 2012
COLLABORATION

During this thesis, I had the following collaborations. Some of them help me on this thesis and other on some parts. Special and many thanks for taking the time, despite your occupation, for following this research work. You are very KIND, OPEN-MINDED and COOL. I am very grateful.

Virginie Wiets
Research engineer at ONERA - the French Aerospace Lab
Information Modeling and Processing Departement
Head of Critical Systems Engineering Research Unit.

Subash Shankar
Professor of Software Engineering
Department of Computer Science, Hunter College and the Graduate Center
City University of New York (CUNY).

John-Jules Ch. Meyer
Professor and Head of Intelligent Systems group
Department of Information and Computing Sciences
Universiteit Utrecht, The Netherlands.

Hans van Ditmarsch
Professor in Logic
Department of Logic
University of Sevilla, Spain.

Pratt & Whitney Canada
An Aerospace Company
Department of E-Business Innovation
Overhaul and Engineering group.
RÉSUMÉ

Un objectif principal du génie logiciel est de pouvoir produire des logiciels complexes, de grande taille et fiables en un temps raisonnable. La technologie orientée objet (OO) a fourni de bons concepts et des techniques de modélisation et de programmation qui ont permis de développer des applications complexes tant dans le monde académique que dans le monde industriel. Cette expérience a cependant permis de découvrir les faiblesses du paradigme objet (par exemples, la dispersion de code et le problème de traçabilité). La programmation orientée aspect (OA) apporte une solution simple aux limitations de la programmation OO, telle que le problème des préoccupations transversales. Ces préoccupations transversales se traduisent par la dispersion du même code dans plusieurs modules du système ou l’emmêlement de plusieurs morceaux de code dans un même module. Cette nouvelle méthode de programmer permet d’implémenter chaque problématique indépendamment des autres, puis de les assembler selon des règles bien définies. La programmation OA promet donc une meilleure productivité, une meilleure réutilisation du code et une meilleure adaptation du code aux changements. Très vite, cette nouvelle façon de faire s’est vue s’étendre sur tout le processus de développement de logiciel en ayant pour but de préserver la modularité et la traçabilité, qui sont deux propriétés importantes des logiciels de bonne qualité.

Cependant, la technologie OA présente de nombreux défis. Le raisonnement, la spécification, et la vérification des programmes OA présentent des difficultés d’autant plus que ces programmes évoluent dans le temps. Par conséquent, le raisonnement modulaire de ces programmes est requis sinon ils nécessiteraient d’être réexamínés au complet chaque fois qu’un composant est changé ou ajouté. Il est cependant bien connu dans la littérature que le raisonnement modulaire sur les programmes OA est difficile vu que les aspects appliqués changent souvent le comportement de leurs composantes de base [47]. Ces mêmes difficultés sont présentes au niveau des phases de spécification et de vérification du processus de développement des logiciels. Au meilleur de nos connaissances, la spécification modulaire et la vérification modulaire sont faiblement couvertes et constituent un champ de recherche très intéressant. De même, les interactions entre aspects est un sérieux problème dans la communauté des aspects. Pour faire face à ces problèmes, nous avons choisi d’utiliser la théorie des catégories et les techniques des spécifications algébriques.

Pour apporter une solution aux problèmes ci-dessus cités, nous avons utilisé les
travaux de Wiels [110] et d’autres contributions telles que celles décrites dans le livre [25]. Nous supposons que le système en développement est déjà décomposé en aspects et classes. La première contribution de notre thèse est l’extension des techniques des spécifications algébriques à la notion d’aspect. Deuxièmement, nous avons défini une logique, $L_A$, qui est utilisée dans le corps des spécifications pour décrire le comportement de ces composantes. La troisième contribution consiste en la définition de l’opérateur de tissage qui correspond à la relation d’interconnexion entre les modules d’aspect et les modules de classe. La quatrième contribution concerne le développement d’un mécanisme de prévention qui permet de prévenir les interactions indésirables dans les systèmes orientés aspect.

**Mots clés:** Génie Logiciel; Développement Orienté Aspect; Spécification Formelle; Vérification Formelle; Interaction Aspect; Mécanisme de Prévention; Raisonnement Modulaire; Modularité.
ABSTRACT

One of the main goals of software engineering is to enable the construction of large, complex and reliable software in timely fashion. Object-oriented (OO) technology has provided modeling and programming principles and techniques that allow developing complex software systems both in academic and industrial areas. In return, experience gained in OO system development has allowed discovering some limitations of object technology (e.g., code scattering and poor traceability problems). Aspect Oriented (AO) Technology is a post-object-oriented technology emerged to overcome limitations of Object Oriented (OO) Technology, such as the crosscutting concern problem. Crosscutting concerns are scattered and tangled concerns. Major goals of Aspect Oriented Programming (AOP) include improving modularity, cohesion, and overall software quality. Aspect Oriented Programming results in the evolution of programming activities to full-blown software engineering processes, to preserve modularity and traceability, which are two important properties of high-quality software.

Yet, there are also many challenges in AO Technology. Reasoning, specification, and verification of AO programs present unique challenges especially as such programs evolve over time. Consequently, modular reasoning of such programs is highly attractive as it enables tractable evolution, otherwise necessitating that the entire program be reexamined each time a component is changed or is added. It is well known in the literature, however, that modular reasoning about AO programs is difficult due to the fact that the aspects applied often alter the behavior of the base components [47]. The same modular reasoning difficulties are also present in the specification and verification phases of software development process. To the best of our knowledge, AO modular specification and verification is a weakly covered subject and constitutes an interesting open research field. Also, aspect interaction is a major concern in the aspect-oriented community. To deal with these problems, we choose to use category theory and algebraic specification techniques.

To achieve the above thesis goals, we use the work of Wiel <110> and other contributions such as the one described in <25>. We assume at the beginning that the system under development is already decomposed into aspect and class components. The first contribution of our thesis is the extension of the algebraic specification technique to the notion of aspect. Secondly, we define a logic, $L_A$ that is used in specification bodies to describe the behavior of these components. The third contribution concerns the defini-
tion of the weaving operator corresponding to the weaving interconnection relationship between aspect modules and class modules. The fourth contribution consists of the design of a prevention policy that is used to prevent or avoid undesirable aspect interactions in aspect-oriented systems.

Keywords: Software Engineering; Aspect Oriented Development; Formal Specification; Formal Verification; Aspect Interaction; Prevention Policy; Modular Reasoning; Modularity.
CONTENTS

RÉSUMÉ ................................................................. iv

ABSTRACT ............................................................. vi

CONTENTS ............................................................. viii

LIST OF TABLES ....................................................... xiii

LIST OF FIGURES .................................................... xiv

DEDICATION .......................................................... xviii

ACKNOWLEDGMENTS ................................................ xix

CHAPTER 1: INTRODUCTION ........................................ 1
  1.1 Software Engineering ......................................... 1
  1.2 Formal Methods ............................................... 2
  1.3 Problem Statement and Objective ........................... 4
  1.4 Abstract View of our Approach .............................. 8
  1.5 Contributions .................................................. 11
    1.5.1 Extension of Algebraic Specification Technique to
          Aspect Orientation ........................................... 11
    1.5.2 Logic $\mathcal{L}_A$ ........................................... 12
    1.5.3 Weaving Algorithm ...................................... 12
    1.5.4 Prevention Policy ...................................... 12
  1.6 Dissertation Organization .................................. 13

CHAPTER 2: BACKGROUND ......................................... 14
  2.1 Category Theory ............................................. 14
  2.2 Universal Constructions in Category Theory ............... 18
    2.2.1 Initial and Terminal Objects ........................... 19
    2.2.2 Sum and Product ...................................... 19
    2.2.3 Pushouts and Pullbacks ............................... 20
    2.2.4 Limits and Colimits ................................... 24
2.3 Many-Sorted Sets and Partial Order .......................... 26
  2.3.1 Many-Sorted Sets ......................................... 26
  2.3.2 Partial Order .............................................. 27
2.4 Work of Ehrig and Mahr ........................................ 28
  2.4.1 Algebraic Specification ................................. 28
  2.4.2 Parameterized Specification .......................... 30
  2.4.3 Module Specification ................................. 31
2.5 Conclusion .................................................. 35

CHAPTER 3: STATE OF THE ART IN ASPECT-ORIENTED MODELLING AND VERIFICATION APPROACHES ........ 36
  3.1 Aspect Oriented Technology ................................. 36
  3.2 Comparative Study of the Main Aspect Oriented Modeling Approaches ......................... 42
    3.2.1 Evaluation Criteria ........................................ 42
    3.2.2 Brief Study of Aspect Oriented Modeling Approaches .................. 44
  3.3 Comparative Study of the Main Aspect Oriented Verification and Validation Approaches .................... 50
    3.3.1 Evaluation Criteria and Selection Criteria .................... 51
    3.3.2 Brief Study of Aspect-Oriented Verification Approaches ........ 52
  3.4 Thesis of Wiels .............................................. 57
  3.5 Conclusion .................................................. 59

CHAPTER 4: LOGIC $L_A$ ........................................... 61
  4.1 Introduction .................................................. 61
  4.2 Syntax of $L_A$ ................................................ 63
  4.3 Semantics of $L_A$ ............................................. 65
  4.4 Satisfiability and Validity .................................... 73
  4.5 Axiom System and Inference Rules .......................... 73
  4.6 Soundness and Completeness ................................. 75
    4.6.1 Soundness ................................................ 75
    4.6.2 Completeness ............................................. 80
  4.7 Related Work ................................................ 87
  4.8 Classical Paradoxes ......................................... 88
    4.8.1 Ross’s Paradox ........................................... 88
    4.8.2 Chisholm’s Paradox .................................... 89
4.8.3 The Free Choice Permission Paradox ......................... 90
4.8.4 The Gentle Murderer Paradox ............................. 90
4.9 Conclusion .................................................. 91

CHAPTER 5: SYNTAXES AND SEMANTICS OF OUR APPROACH . . 92
5.1 Introduction ................................................. 92
5.2 Specifications .............................................. 92
  5.2.1 Concrete Syntax of Specifications ......................... 92
  5.2.2 Abstract Syntax of Specifications ....................... 105
  5.2.3 Specification Morphism .................................. 111
5.3 Module specification ........................................ 116
  5.3.1 Module specification .................................... 116
  5.3.2 Module morphism ...................................... 116
5.4 Category of Module Specifications ............................ 116
  5.4.1 Category of Specifications ............................... 118
  5.4.2 Category of Module Specifications .................... 121
5.5 Semantics .................................................. 125
  5.5.1 Semantics of Specifications ............................ 126
  5.5.2 Semantics of Module Specifications ................. 132
5.6 Conclusion .................................................. 136

CHAPTER 6: MODULE INTERCONNECTION: WEAVING ALGORITHM .......................... 137
6.1 Introduction ................................................. 137
6.2 Aspect-Class Relationship ................................ 138
  6.2.1 Weaving ............................................... 138
  6.2.2 Relationship between Aspect and Class Modules ...... 140
6.3 Interconnection of Modules ................................ 142
  6.3.1 Modules Composition .................................. 143
  6.3.2 Product of Modules .................................... 144
  6.3.3 Union of Modules ...................................... 145
  6.3.4 Actualization of Module ............................... 148
  6.3.5 Extension of Module .................................. 149
6.4 Weaving Operator .......................................... 150
  6.4.1 Three Way Merging .................................... 152
### CHAPTER 7: ASPECT PREVENTION MECHANISM

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Introduction</td>
<td>167</td>
</tr>
<tr>
<td>7.2 A fault model for AOPs</td>
<td>168</td>
</tr>
<tr>
<td>7.2.1 Incorrect Aspect Precedence</td>
<td>169</td>
</tr>
<tr>
<td>7.2.2 Incorrect Strength in Pointcut Patterns</td>
<td>169</td>
</tr>
<tr>
<td>7.2.3 Failure to Establish Expected Post-Conditions</td>
<td>170</td>
</tr>
<tr>
<td>7.2.4 Failure to Preserve State Invariants</td>
<td>171</td>
</tr>
<tr>
<td>7.2.5 Incorrect Focus of Control Flow</td>
<td>171</td>
</tr>
<tr>
<td>7.2.6 Incorrect Changes in Control Dependencies</td>
<td>172</td>
</tr>
<tr>
<td>7.3 Aspect Prevention Mechanism</td>
<td>172</td>
</tr>
<tr>
<td>7.3.1 Model of Incorrect Aspect Precedence</td>
<td>172</td>
</tr>
<tr>
<td>7.3.2 Model of &quot;Incorrect Strength in Pointcut Patterns&quot;</td>
<td>176</td>
</tr>
<tr>
<td>7.3.3 Model of &quot;Failure to Establish Expected Post-Conditions&quot;</td>
<td>177</td>
</tr>
<tr>
<td>7.3.4 Model of &quot;Failure to Preserve State Invariants&quot;</td>
<td>179</td>
</tr>
<tr>
<td>7.3.5 Examples</td>
<td>179</td>
</tr>
<tr>
<td>7.4 Verification</td>
<td>183</td>
</tr>
<tr>
<td>7.5 Conclusion and Future Work</td>
<td>184</td>
</tr>
</tbody>
</table>

### CHAPTER 8: CASE STUDY

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Project Description</td>
<td>186</td>
</tr>
<tr>
<td>8.2 Partial Modular Specification of the MRO System</td>
<td>187</td>
</tr>
<tr>
<td>8.2.1 Aspect Identification</td>
<td>196</td>
</tr>
<tr>
<td>8.2.2 Specification of the Case Study with the Modules</td>
<td>211</td>
</tr>
<tr>
<td>8.3 Weaving</td>
<td>234</td>
</tr>
<tr>
<td>8.3.1 Weaving of CoordinationAspect into the Mechanic Class</td>
<td>234</td>
</tr>
<tr>
<td>8.3.2 Weaving of AuthenticationAspect into $MOD_{w1}$</td>
<td>239</td>
</tr>
<tr>
<td>8.4 Conclusion</td>
<td>249</td>
</tr>
</tbody>
</table>
CHAPTER 9: CONCLUSION ................................................. 250

9.1 Contributions ......................................................... 250
  9.1.1 Extension of Algebraic Specification Technique to Aspect Ori-
        entation .................................................. 250
  9.1.2 Logic $L_A$ .................................................. 251
  9.1.3 Weaving Algorithm .......................................... 252
  9.1.4 Prevention Policy ............................................ 252
  9.1.5 Application to MRO Applications ......................... 253

9.2 Future Research Directions ........................................ 253

BIBLIOGRAPHY ............................................................. 255
LIST OF TABLES

3.I Summary results of AOM approaches analysis .......................... 50
3.II Summary results of aspect-oriented VV approaches analysis .... 57
8.I Actor activities: Part 1 ...................................................... 190
8.II Actor activities: Part 2 ..................................................... 191
8.III Use Case Description: Part 1 ........................................ 192
8.IV Use Case Description: Part 2 ........................................ 193
8.V Use Case Description: Part 3 ........................................... 194
8.VI Authentication Aspect Weaving Information for Mechanic and AISDatabase .......................................................... 197
8.VII Data Origin Authentication Aspect Weaving Information for Mechanic and AISDatabase ............................................. 199
8.VIII Role-Based Access Control Aspect Weaving Information for AISDatabase ......................................................... 203
8.IX Log for Audit Aspect Weaving Information for Mechanic and AISDatabase .......................................................... 208
8.X File Versioning Aspect Weaving Information for AISDatabase .. 211
# LIST OF FIGURES

1.1 Example of a Modular Verification ............................................. 8
1.2 Abstract View of our Approach .................................................. 11
2.1 Specification of semigroup ......................................................... 29
2.2 Specification of boolean values .................................................. 30
2.3 Parameterized specification of ADT String(data) ...................... 32
2.4 Module specification .............................................................. 33
2.5 Module specification BOOL ....................................................... 34
2.6 Module morphism ..................................................................... 35
3.1 Example of a pointcut with parameters .................................... 37
3.2 Object-oriented model for a simple figure editor system ............ 39
3.3 Java code of class Point ........................................................... 39
3.4 Java code of class Line ............................................................. 40
3.5 Java code of class Point after SoC ............................................ 40
3.6 Java code of class Line after SoC ............................................. 41
3.7 Aspect DisplayUpdating in AspectJ .......................................... 41
3.8 Parallel of our approach with Wiels approach ......................... 60
5.1 Pattern of an ADT Specification .............................................. 94
5.2 Specification of ADT String ..................................................... 95
5.3 Pattern of a CLASS specification ............................................. 97
5.4 Societal life of aspects and classes .......................................... 98
5.5 Specification of Invoice Class ................................................. 99
5.6 Pattern of an ASPECT Specification ....................................... 100
5.7 Verification Aspect Specification .......................................... 103
5.8 Log ASPECT Specification .................................................... 104
5.9 Module specification .............................................................. 134
5.10 Model of a Module specification ........................................... 135
6.1 The Weaving Process ............................................................. 139
6.2 Hello word class ................................................................. 139
6.3 Hello word aspect ............................................................... 139
6.4 Hello word augmented system .............................................. 140
6.5 The pointcut association: relating aspects to classes ............... 141
6.6 Modules composition ............................................. 143
6.7 Modules Product ................................................... 145
6.8 Union of two Module Specifications .............................. 147
6.9 Actualization of a Module Specification ......................... 149
6.10 Extension of a Module Specification ............................ 150
6.11 Module specification of Class Point, Part 1 ..................... 157
6.12 Module specification of Class Point, Part 2 ..................... 158
6.13 Module specification of Class Point, Part 3 ..................... 158
6.14 Module specification of Class Point, Part 4 ..................... 158
6.15 Module specification of Aspect DisplayUpdating, Part 1 ....... 159
6.16 Module specification of Aspect DisplayUpdating, Part 2 ....... 159
6.17 Module specification of Aspect DisplayUpdating, Part 3 ....... 159
6.18 Module specification of Aspect DisplayUpdating, Part 4 ....... 160
6.19 Module specification of Aspect DisplayUpdating, Part 5 ....... 160
6.20 Module specification of Class Point: Part 1 ..................... 161
6.21 Module specification of Class Point: Part 2 ..................... 161
6.22 Module specification of Class Point: Part 3 ..................... 161
6.23 Module specification $M_U$, Part 1 ............................. 162
6.24 Module specification $M_U$, Part 2 ............................. 163
6.25 Module specification $M_U$, Part 3 ............................. 163
6.26 Module specification $MOD_w$, Part 1 ......................... 164
6.27 Module specification $MOD_w$, Part 2 ........................ 164
6.28 Module specification $MOD_w$, Part 3 ........................ 164
6.29 Module specification $MOD_w$, Part 4 ........................ 165
6.30 Module specification $MOD_w$, Part 5 ........................ 165

7.1 Types of faults in pointcut descriptors ............................... 171
7.2 The Employee Class and its superimposed aspects .................. 173
7.3 The class/aspect diagram for card purchasing ..................... 181

8.1 Use Case Model of MRO Application .............................. 189
8.2 Class Diagram of MRO Application .............................. 195
8.3 Static View of the Authentication Aspect ......................... 196
8.4 Data origin authentication aspect with two base classes .......... 199
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.5</td>
<td>Static View of Role-Based Access Control Aspect.</td>
<td>202</td>
</tr>
<tr>
<td>8.6</td>
<td>Static View of Log for Audit Aspect.</td>
<td>204</td>
</tr>
<tr>
<td>8.7</td>
<td>Static View of Log for File Versioning Aspect.</td>
<td>209</td>
</tr>
<tr>
<td>8.8</td>
<td>Module specification of Class Mechanic, Part 1</td>
<td>211</td>
</tr>
<tr>
<td>8.9</td>
<td>Module specification of Class Mechanic, Part 2</td>
<td>212</td>
</tr>
<tr>
<td>8.10</td>
<td>Module specification of Class Mechanic, Part 3</td>
<td>212</td>
</tr>
<tr>
<td>8.11</td>
<td>Module specification of Class Mechanic, Part 4</td>
<td>212</td>
</tr>
<tr>
<td>8.12</td>
<td>Module specification of Class AISDatabase, Part 1</td>
<td>213</td>
</tr>
<tr>
<td>8.13</td>
<td>Module specification of Class AISDatabase, Part 2</td>
<td>213</td>
</tr>
<tr>
<td>8.14</td>
<td>Module specification of Class AISDatabase, Part 3</td>
<td>213</td>
</tr>
<tr>
<td>8.15</td>
<td>Module specification of Class AISDatabase, Part 4</td>
<td>214</td>
</tr>
<tr>
<td>8.16</td>
<td>Module specification of Aspect AuthenticationAspect, Part 1</td>
<td>214</td>
</tr>
<tr>
<td>8.17</td>
<td>Module specification of Aspect AuthenticationAspect, Part 2</td>
<td>215</td>
</tr>
<tr>
<td>8.18</td>
<td>Module specification of Aspect AuthenticationAspect, Part 3</td>
<td>215</td>
</tr>
<tr>
<td>8.19</td>
<td>Module specification of Aspect AuthenticationAspect, Part 4</td>
<td>215</td>
</tr>
<tr>
<td>8.20</td>
<td>Module specification of Aspect DOAAspect, Part 1</td>
<td>216</td>
</tr>
<tr>
<td>8.21</td>
<td>Module specification of Aspect DOAAspect, Part 2</td>
<td>216</td>
</tr>
<tr>
<td>8.22</td>
<td>Module specification of Aspect DOAAspect, Part 3</td>
<td>217</td>
</tr>
<tr>
<td>8.23</td>
<td>Module specification of Aspect DOAAspect, Part 5</td>
<td>217</td>
</tr>
<tr>
<td>8.24</td>
<td>Module specification of Aspect DOAAspect, Part 41</td>
<td>218</td>
</tr>
<tr>
<td>8.25</td>
<td>Module specification of Aspect DOAAspect, Part 42</td>
<td>219</td>
</tr>
<tr>
<td>8.26</td>
<td>Module specification of Aspect RBACAspect, Part 1</td>
<td>219</td>
</tr>
<tr>
<td>8.27</td>
<td>Module specification of Aspect RBACAspect, Part 2</td>
<td>220</td>
</tr>
<tr>
<td>8.28</td>
<td>Module specification of Aspect RBACAspect, Part 3</td>
<td>220</td>
</tr>
<tr>
<td>8.29</td>
<td>Module specification of Aspect RBACAspect, Part 41</td>
<td>221</td>
</tr>
<tr>
<td>8.30</td>
<td>Module specification of Aspect RBACAspect, Part 42</td>
<td>222</td>
</tr>
<tr>
<td>8.31</td>
<td>Module specification of Aspect RBACAspect, Part 5</td>
<td>222</td>
</tr>
<tr>
<td>8.32</td>
<td>Module specification of Aspect LogAspect, Part 1</td>
<td>223</td>
</tr>
<tr>
<td>8.33</td>
<td>Module specification of Aspect LogAspect, Part 2</td>
<td>224</td>
</tr>
<tr>
<td>8.34</td>
<td>Module specification of Aspect LogAspect, Part 31</td>
<td>225</td>
</tr>
<tr>
<td>8.35</td>
<td>Module specification of Aspect LogAspect, Part 32</td>
<td>226</td>
</tr>
<tr>
<td>8.36</td>
<td>Module specification of Aspect LogAspect, Part 4</td>
<td>226</td>
</tr>
<tr>
<td>8.37</td>
<td>Module specification of Aspect VersionningAspect, Part 1</td>
<td>227</td>
</tr>
<tr>
<td>8.38</td>
<td>Module specification of Aspect VersionningAspect, Part 2</td>
<td>227</td>
</tr>
<tr>
<td>8.39</td>
<td>Module specification of Aspect VersionningAspect, Part 3</td>
<td>228</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.40</td>
<td>Module specification of Aspect VersionningAspect, Part41</td>
<td>229</td>
</tr>
<tr>
<td>8.41</td>
<td>Module specification of Aspect VersionningAspect, Part42</td>
<td>230</td>
</tr>
<tr>
<td>8.42</td>
<td>Module specification of Aspect VersionningAspect, Part 5</td>
<td>230</td>
</tr>
<tr>
<td>8.43</td>
<td>Module specification of Aspect CoordinationAspect, Part1</td>
<td>232</td>
</tr>
<tr>
<td>8.44</td>
<td>Module specification of Aspect CoordinationAspect, Part2</td>
<td>232</td>
</tr>
<tr>
<td>8.45</td>
<td>Module specification of Aspect CoordinationAspect, Part3</td>
<td>233</td>
</tr>
<tr>
<td>8.46</td>
<td>Module specification of Aspect CoordinationAspect, Part 4</td>
<td>234</td>
</tr>
<tr>
<td>8.47</td>
<td>Module specification of $MOD_{S1}$, Part1</td>
<td>234</td>
</tr>
<tr>
<td>8.48</td>
<td>Module specification of $MOD_{S1}$, Part 2</td>
<td>235</td>
</tr>
<tr>
<td>8.49</td>
<td>Module specification of the weaving of Mechanic with CoordinationAspect, Part1</td>
<td>235</td>
</tr>
<tr>
<td>8.50</td>
<td>Module specification of the weaving of Mechanic with CoordinationAspect, Part2</td>
<td>236</td>
</tr>
<tr>
<td>8.51</td>
<td>Module specification of the weaving of Mechanic with CoordinationAspect, Part3</td>
<td>237</td>
</tr>
<tr>
<td>8.52</td>
<td>Module specification of the weaving of Mechanic with CoordinationAspect, Part4</td>
<td>238</td>
</tr>
<tr>
<td>8.53</td>
<td>Module specification of the weaving of Mechanic with CoordinationAspect, Part 5</td>
<td>239</td>
</tr>
<tr>
<td>8.54</td>
<td>Module specification of $MOD_{S2}$, Part 1</td>
<td>239</td>
</tr>
<tr>
<td>8.55</td>
<td>Module specification of $MOD_{S2}$, Part 2</td>
<td>240</td>
</tr>
<tr>
<td>8.56</td>
<td>Module specification of $MOD_{S2}$, Part 3</td>
<td>240</td>
</tr>
<tr>
<td>8.57</td>
<td>Module specification of the $MOD_{u2}$, Part1</td>
<td>241</td>
</tr>
<tr>
<td>8.58</td>
<td>Module specification of the $MOD_{u2}$, Part1</td>
<td>241</td>
</tr>
<tr>
<td>8.59</td>
<td>Module specification of $MOD_{u2}$, Part2</td>
<td>242</td>
</tr>
<tr>
<td>8.60</td>
<td>Module specification of $MOD_{u2}$, Part3</td>
<td>243</td>
</tr>
<tr>
<td>8.61</td>
<td>Module specification of $MOD_{u2}$, Part4</td>
<td>244</td>
</tr>
<tr>
<td>8.62</td>
<td>Module specification of the $MOD_{u2}$, Part 5</td>
<td>245</td>
</tr>
<tr>
<td>8.63</td>
<td>Module specification of the $MOD_{w2}$, Part1</td>
<td>245</td>
</tr>
<tr>
<td>8.64</td>
<td>Module specification of $MOD_{w2}$, Part2</td>
<td>246</td>
</tr>
<tr>
<td>8.65</td>
<td>Module specification of $MOD_{w2}$, Part3</td>
<td>247</td>
</tr>
<tr>
<td>8.66</td>
<td>Module specification of $MOD_{w2}$, Part4</td>
<td>248</td>
</tr>
<tr>
<td>8.67</td>
<td>Module specification of the $MOD_{w2}$, Part 5</td>
<td>248</td>
</tr>
</tbody>
</table>
Dedicate

To

My father Sabas Alohou
My mother Sabine Alohou
My brother Étienne
My sisters

I dedicate this thesis.
ACKNOWLEDGMENTS

I would like to express my gratitude to my advisors, Michel Boyer and Michael Barr, for their support, their help, their understanding and their disponibility during this thesis. I appreciated a lot. Special and many thanks to Houari Sahraoui for his continuous, practical and pragmatical advices during this thesis. They trusted me and this is very important to me.

I would like also to thank the members of jury for accepting to evaluate this thesis.

I thank Aouni A. Lakis, head of Institut d’Inovation et de Conception en Aérospatiale de Polytechnique (IICAP), Gladys Daoud, coordinator of IICAP, Isabelle Daoust, adjunct of associate dean for graduate studies and external affairs of Faculté des arts et des sciences, Jenson Jane, ex associate dean for graduate studies and external affairs of Faculté des arts et des sciences, Jean-Yves Potvin, responsible for graduate studies of department of computer science and operation research, and Thierry Paul, project manager at Bureau de la recherche et centre de développement technologique, for allowing me to do the training program at Pratt & Whitney Canada.

I thank Patrice Marcotte, head of department of computer science and operation research for his support during this thesis. I thank Prof. Yann-Gaël Guéhéneuc for his help and moral support during my synthesis examination.

I thank Fonds de Recherche du Québec Nature et technologies (FQRNT) and the department of computer science and operation research for their financial support.

I own a special note of gratitude to my friends, Pascal N’dinga, Clément Yelou, Guy Houns, Hervé Koussé, Floris Randolph and Ulrich Hounyo for always being there when I needed them and their moral support. Thank to my friend and colleague, the small black Jean-Luc Victor for his jokes. Thank to any one who supports me in one way or another during this thesis period.

My deepest gratitude goes to my parents, Sabas Alohou and Sabine Alohou, for believing in me and encouraging me. I also own a special note of gratitude to my sister Faustine Alohou for her unconditional friendship and moral support.
CHAPTER 1

INTRODUCTION

1.1 Software Engineering

We can’t run the modern world without software. National infrastructures and utilities are controlled by computer-based systems and most electrical products include a computer and controlling software. Industrial manufacturing and distribution is completely computerized, as is the financial system. Entertainment, including the music industry, computer games, and film and television, is software intensive. Therefore, software engineering is essential for the functioning of national and international societies.

Many definitions have been proposed, but, in general, software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use [98]. Software Engineering is a multifaceted and expanding topic. It aims to provide theories, methods and tools to tackle the complexity of software systems, from development to maintenance. Its complexity is made even more severe today by rapid advances in technology, the pervasiveness of software in all areas of society, and the globalization of software development.

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of steps that leads to the production of a software product. Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of analyzing requirements, specifying, designing, implementing, and validating and verifying, and maintaining software systems. We are aware that there is no commonly accepted model for the software development process from a given problem via specification and design to an efficient version of the software system. But the analysis phase takes the informal requirements as its input and outputs a specification. The design phase uses and refines iteratively this output specification of the analysis phase to produce a detailed specification. The implementation phase uses this detailed specification to produce the final product, which will be validated and verified. The code is tested at various levels in software testing. Software verification is a broader and more complex discipline of software engineering whose goal is to assure that software fully satisfies all the expected requirements. This
verification could be spread over other phases. Testing is a particular case of verification. Software verification is often confused with software validation. The difference between verification and validation:

- Software verification asks the question, "Are we building the product right?"; that is, does the software conform to its specification.
- Software validation asks the question, "Are we building the right product?"; that is, is the software doing what the user really requires.

What happens during the rest of the software’s life: changes, correction, additions, moves to a different computing platform and more. This is often the longest of the stages. This is the maintenance phase.

The purpose of software engineering is to reduce software development time and cost, to discipline the phases of software process and ensure the quality of them. Development of increasingly complex systems has created a need for improved specification and verification techniques. Specification is the process of describing a system and its desired properties. Formal specification uses a language with a mathematically defined syntax and semantics. The process of specification is the act of writing things down precisely. The main benefit in so doing is important, gaining a deeper understanding of the system being specified. It is through this specification process that developers uncover design flaws, inconsistencies, ambiguities, and incompleteness. A tangible product of this process, however, is an artifact that can itself be formally analyzed, for example, checked to be internally consistent or used to derive other properties of the specified system. The specification is a useful communication device between customer and designer, between designer and implementor, and between implementor and tester. It serves as a companion document to the system’s source code, but at a higher level of description [16].

The work we present here belongs to the discipline of Software Engineering, at the specification and verification phases.

1.2 Formal Methods

Hardware and software systems will inevitably grow in scale and functionality. Because of this increase in complexity, the likelihood of subtle errors is much greater. Moreover, some of these errors may cause catastrophic loss of money, time, or even hu-
man life. These systems are present in all domains (medicine, banking systems, telecommunication, spatial, nuclear, etc.) and particularly in critical systems (Ariane 5, Airbus A320, TGV) for which we must ensure their correct functionality. The classical methods used in software engineering are not adequate for such systems. A major goal of software engineering is to enable developers to construct systems that operate reliably despite this complexity. One way of achieving this goal is by using formal methods. Formal Methods consist of a set of techniques and tools based on mathematical modeling and formal logic that are used to specify and verify requirements and designs for computer systems and software. The use of formal methods on a project can assume various forms, ranging from occasional mathematical notation embedded in English specifications, to fully formal specifications using specification languages with a precise semantics. At their most rigorous, formal methods involve computer-assisted proofs of key properties regarding the behavior of the system [66].

Use of formal methods does not a priori guarantee correctness. However, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected. The following are some of the benefits realizable from effective applications of formal methods [66]:

- Formal methods (theorem proving, model checking, for example) help find defects; as evidence of this, when applied to high-quality software systems, formal methods have found defects that went undetected during extensive testing. The non-exhaustive nature of testing ensures that complex systems will always have scenarios which cannot be tested due to practical considerations.

- Formal specifications allow defects in requirements and designs to be detected earlier than they would be otherwise and greatly reduce the incidence of mistakes in interpreting and implementing correct requirements and designs.

- Formalized statements can be analyzed and their consequences calculated in a repeatable manner. The risks of drawing conclusions about a system’s behavior by extrapolating from a finite number of tests often can be avoided by using proof methods based on mathematics. Such methods allow large (potentially infinite) classes of test cases to be fully covered in a finite proof, and they support reasoning that can be checked by colleagues or by machine, with minimal dependence on subjective reasoning.

- Use of formal methods causes more defects to be detected than would otherwise
be the case and in certain circumstances guarantees the absence of certain defects.

Our approach described in this thesis uses formal methods to model and verify aspect-oriented systems.

1.3 Problem Statement and Objective

Object-oriented technology has provided modeling and programming principles and techniques that allow developing complex software systems both in academic and industrial areas. In return, experience gained in object-oriented system development has allowed discovering some limitations of object technology. Aspect-Oriented Programming (AOP) [50] is a new technology for improving comprehension and maintainability of complex programs by localizing behaviors that would otherwise be scattered and tangled. These behaviors are referred to as crosscutting concerns. Scattering is the condition in which a concern is implemented in several non-contiguous places in the program. For example, code that implements a particular security policy is commonly spread over many classes and methods that are responsible for enforcing the policy. Tangling, the dual of scattering, occurs when several concerns overlap at a region in the program text. Traditional Object-Oriented Programming (OOP) cannot control these crosscutting concerns. AOP targets the separation of crosscutting concerns by isolating each crosscutting concern from the core concerns (the rest of code) into modules called aspects. This aspect identification is called the decomposition step.

After this decomposition step, all aspects are composed into their base components by a weaving mechanism to get the whole system. Aspect-Oriented Technology (AOT) provides weaving mechanisms to integrate each aspect into its base component (core concern). The result of the weaving is the complete system. Without the aspect components, the base system will be more maintainable. As such, aspect-oriented programs tend to enjoy plug-n-play-type capabilities where base and/or aspect components may be introduced, removed, and interchanged easily (in [47]). Major goals of AOP include improving of modularity, cohesion, and overall software quality. AOP enables the modular implementation of crosscutting concerns. AOP results in the evolution of programming activities to full-blown software engineering processes, to preserve modularity and traceability, which are two important properties of high-quality software. Modularity means that we want to split larger specifications into smaller parts, where each part is well-defined in itself but leaves structure and semantics of some subparts open [25]. Requirements or concern traceability is defined as the ability to describe and follow the life
of a requirement, in both a forward and backward direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases) [102]. Aspect-oriented software development (AOSD) is a post object-oriented technology contributing to better modularization by providing mechanisms to localize cross-cutting concerns during software development process. AOP is thus directly supported by several modern languages such as AspectJ [50].

Yet, there are also many challenges in Aspect-Oriented Technology. We have identified two main problems in the literature for which we propose solutions in our thesis:

**Problem 1:** Reasoning, specification, and verification of aspect-oriented programs present unique challenges especially as such programs evolve over time. Consequently, modular reasoning of such programs is highly attractive as it enables tractable evolution, otherwise necessitating that the entire program be reexamined each time a component is changed or is added. It is well known, however, that modular reasoning about aspect-oriented programs is difficult due to the fact that the applied aspects often alter the behavior of the base components [47]. The same modular reasoning difficulties are also present in the specification and verification phases of software development process. To the best of our knowledge, aspect-oriented modular specification and verification is a weakly covered subject and constitutes an interesting open research field. AOP modularity concepts are not present at the specification and verification level for aspect-oriented systems. This problem has been pointed out in the paper [103].

As defined in [49], modular reasoning means being able to make decisions about a module while looking only at its implementation, its interface and the interfaces of modules referenced in its implementation or interface. Expanded modular reasoning means also consulting the implementations of referenced modules, and global reasoning means having to examine all the modules in the system or subsystem.

**Problem 2:** An increase in software quality does not, however, imply that programmers will stop making mistakes and that Aspect Oriented software will be bug free. Aspect interaction is one of the main concerns in the aspect-oriented community [23]. The aspect interaction problem can be presented as follows [96]:

- Let $P_1$ be a property satisfied by aspect $A$. 
• Let $P_2$ be a property satisfied by the composition of the base system and $n \geq 1$ aspects (SYS); $n$ is a positive integer. $A$ is not yet composed with SYS.

• Both $P_1$ and $P_2$ should remain satisfied by the composition of $A$ and SYS; if not, then, there is an interaction problem.

The interaction between aspects and classes may introduce a variety of bug hazards into the system. Detection and resolution of undesirable aspect interactions is an important open research field. Care must be taken that the woven behavior of two concerns satisfies both existing critical correctness properties of the behavior of each individual concern and new desired correctness properties of the woven system. Otherwise, there is undesirable interactions. Most aspect-oriented verification approaches are based on a detection and correction strategy. Although these detection approaches are relevant for aspect-oriented software reliability, we believe that they are time consuming and costly.

It is a good thing to detect and correct system failures, but it is better to first prevent them and it is a prevention policy, to be integrated at the specification phase, that we will be advocating. We believe indeed that this will make the verification phase much faster and cheaper. We can convince ourselves by making an analogy to medicine, where experts and governments prefer to place more emphasis on measures to prevent disease. The same analogy could be observed in avionics where MRO (maintenance, repair, overhaul) activities put more emphasis on preventive actions than corrective actions for flight quality and reliable aircraft.

An aspect fault model developed in [5] lists the main fault types that can arise in aspect-oriented applications due to the aspect integration into the base systems (see chapter 7 for the description of these fault types). Existing verification approaches can detect only one or two of these fault types. To the best of our knowledge, there are no existing methods or tools capable of taking all of them into account. An aspect verification approach survey [46] also points out this. There is a lack of efficient mechanisms for dealing with aspect interactions, and which can take most of these aspect fault types into account. We introduce a logic $L_A$ (see chapter 4) that will help us specify system components with a prevention mechanism. This prevention mechanism will prevent most of the undesirable aspect interactions characterized by the fault classes developed in [5]. We use this logic to describe the behavior of components, and also the societal life of these
components. In our approach, to deal with aspect interaction problems, we adopt both preventive and corrective strategies. A preventive strategy is based on preventing a non-conformant event in the future. A corrective strategy is based on a non-conformant event that has happened in the past. Corrective action will be taken at the verification phase when necessary, i.e., after an interaction problem has occurred. Besides, there is also a need of modularity concept at the aspect-oriented verification phase to bypass some drawbacks of model checking and theorem prover techniques.

Our goal is to resolve these two problems quoted above. In other words, our thesis goals are:

- **Import the AOP modularity concepts into the formal specification and verification level for aspect-oriented systems.**

- **Resolve at the specification and verification phases the undesirable aspect interactions characterized by the fault types quoted above.**

Precisely, our goal is to develop a formal framework for the specification and the verification of aspect-oriented systems, allowing modular reasoning due to the benefits of this later. To help reasoning about aspect-oriented systems, the use of formal methods becomes desirable. Formal methods are essential to support quality, modifiability and reusability by formal concepts for data abstraction and modularity [26]. We also intend to build a prevention mechanism (a set of properties) whose role is to prohibit undesirable interactions.

We believe that category theory and algebraic specification can help us achieve these goals. Category theory is a good and powerful tool for the modularization of system components which can be considered as objects of a category. It introduces the notion of morphism, which can be used as a means to study and to implement interactions within these components. Moreover, it has construction operators that allow to compose in a structural way these components to form the complete system. Researchers have established the usefulness of this mathematic theory for the formalization of software systems [10, 28, 110]. Wiels [3, 61, 110, 111] has used category theory to specify and to verify some existing systems, such as fault-tolerant, telecommunication and avionic systems. To deal with the resolution of undesirable interactions between aspects and base components, we decided to rely on category theory and algebraic specification to model and to verify aspect-oriented systems. By using category theory, we can take
advantage of the structure of a system specification to do the verification task and to infer some desirable properties on the global system. The principle of this modular verification is as follows: for a morphism \( m : \text{Mod}_1 \to \text{Mod}_2 \), if a property \( P \) is true in \( \text{Mod}_1 \), then \( m(P) \) is true in \( \text{Mod}_2 \). That is the principle we will use to do the modular verification. This principle has been used by Sannella and Burstall [90], Fiadeiro and Maibaum [32], and Wiels [110]. For example [61], if we want to prove a property on the module \( \text{ModSystem} \) representing the entire system and we have the following structure or configuration (figure 1.1):

![Figure 1.1: Example of a Modular Verification](image)

This property may be expressed and proved in \( \text{Mod}_3 \), and then translated to \( \text{ModSystem} \). Another property of \( \text{ModSystem} \) may result from the conjunction of two lemmas, each lemma being provable in smallest module (\( \text{Mod}_3 \) and \( \text{Mod}_4 \)). That is one of our main motivation for using category theory. The next section gives an abstract view of our approach.

### 1.4 Abstract View of our Approach

To achieve the above thesis goals, we use the work of Wiels [110] (see section 3.4) and other contributions such as the one described in [25]. As in Wiels approach, there are three levels of system description in our approach:

- a system is described by modules that are interconnected by morphisms and on which composition operations can be performed. A module is an abstraction means to represent a system component. A module can represent aspect or class
components which can use data structures. A morphism is a structure-preserving mapping between two objects in the category context. It is a means to specify an interaction between these two objects.

- a module is composed of four specifications linked by specification morphisms.
- each specification is a logical theory (a signature that gives the vocabulary (attributes and methods) and a set of formulae to describe the behavior).

We assume at the beginning that the system under development is already decomposed into aspect and class components. This decomposition task can be made by any adequate requirement analysis approach. We define a logic, $L_A$ that is used in specification bodies to describe the behavior of these specifications. The logic $L_A$ includes modalities of three other logics:

1. linear temporal logic (LTL) [55] to reason on the time (state);
2. (first-order) dynamic logic (FDL) [43] to reason explicitly on actions or computer programs and properties (LTL cannot explicitly reason on actions);
3. deontic logic [112] to specify the societal life of system components. We know that such system components interact with each other to resolve a common task, and undesirable interactions between these components can arise; we don’t always have what we want (see chapter 4).

After that, we define, by adapting the formalism of Ehrig and Mahr [24], [25] the algebraic specification of aspect, class and abstract data type, and the necessary morphisms between these specifications. Then we define each kind of module specification representing an aspect component, a class component, and an abstract data type. Also, we define module morphisms meaning the relationships between these module specifications. We equip each module with the desirable properties specifying the desirable interactions. These desirable properties contain a prevention mechanism or a policy that prevent or avoid occurrence of undesirable interactions in an aspect-oriented system. This policy has been specified as prescription properties. Prescription means what a system should do. Action prescriptions are means to convey when actions may or must occur (as opposed to just describing the effects of such actions), via the deontic concepts of obligatory, permissible and prohibited action [48].
Then, we build a category whose objects are the underline (class) modules and the aspect modules, and in which the morphisms are the morphisms between these modules. Category concepts (such as the concepts of co-limit) allow to calculate the module that represents the entire system under development (this module is called modular system). We define a weaving algorithm using the concepts of co-limit and the construction operators defined by Ehrig and Mahr [25]. Construction operators allow to compose two modules to obtain a new one.

Afterward, we can use the structure of this diagram to verify, in a modular way, properties describing aspect-aspect and aspect-class interactions. Using this diagram structure, properties can be inferred on the entire system from its components by means of the morphism preservation property. In our approach, to deal with aspect interaction problems, we adopt a preventive and corrective strategies. A preventive strategy is based on preventing a non-conformant event in the future. A corrective strategy is based on a non-conformant event that has happened in the past.

Our prevention mechanism plays the role of a prevention action. Corrective action will be taken at the verification phase when it is necessary, i.e., after an interaction problem has occurred. To the best of our knowledge, aspect-oriented verification approaches are based only on the corrective strategy, i.e., the detection and the resolution of aspect conflicts. Besides, we define of course the semantics of our language (see section 5.5 of chapter 5).

Next, we apply our approach to an industrial case study to validate our framework. This case study is a Pratt & Whitney Canada [2] project on which we worked during a training program we did in this aerospace company. We will extend to aspect concepts (which will be named AMOKA) the tool MOKA developed by the team of Wiels [61, 110] to do some verification tasks. Currently, this tool is in standard ML. We need to rebuild it in an object oriented language such as OCAML, by integrating aspect concepts or we can also use an aspect oriented language such as Aspectual CAML [56], because our approach is aspect-oriented. Note that OCAML is object-oriented version of standard ML. AMOKA will automatically verify the consistency of the categoric modules and some properties on the global system. For those properties which cannot be verified by AMOKA, we will interface AMOKA with a proof tool which can discharge the proofs of these properties. Figure 1.2 shows the schematic view of our approach.
1.5 Contributions

To overcome the problems quoted above, we propose the following contributions:

1.5.1 Extension of Algebraic Specification Technique to Aspect Orientation

Algebraic specification is a formal specification approach that emerged in the mid-70s as a technique to deal with data structures in an implementation-independent manner [24]. The approach was based on specifying data types in a similar way to that used for the study of different mathematical structures (e.g. groups, rings, fields, etc.) in modern algebra. Originally, algebraic specification was intended as a technique for the description of abstract data types. This technique soon grew into a formal specification technique aiming to cover the whole specification phase within the software development
process. Since then, the research efforts have led to extension of this specification technique to object-oriented languages. The first contribution of our thesis is the extension of the algebraic specification technique to the notion of aspect [77, 80, 83, 84, 88]. We define concrete and abstract syntaxes, and a precise semantics of our aspect language. To help reasoning about aspect-oriented systems, the use of formal methods becomes desirable. Formal methods are essential to support quality, modifiability and reusability by formal concepts for data abstraction and modularity [26].

1.5.2 Logic $L_A$

To specify the behavior of software components, we need a logic. This logic has to take into account the societal life of these components which is an important dimension in software system. This societal life has goal to regulate the different interactions between components of a software system. The second contribution is the definition of a logic $L_A$ [78, 82, 86], expressive enough to specify the description and prescription (societal life) behaviors of system components.

1.5.3 Weaving Algorithm

Weaving is the predominant notion of model combination in Aspect Oriented Software Development [34]. Module interconnections form the architectural structure of a modular system. Four types of interconnection have been defined in [25] and are used in [110]. For each of the four relationships, construction operators have been defined by Ehrig and Mahr [25] to realize the interconnection associated. We extend the four types of interconnection by adding a fifth kind, the weaving relation. The process of weaving an aspect to a set of base objects (classes) consists in assembling these entities together to produce the final application extended with the behaviors defined in the aspects. The third contribution concerns the definition of the weaving operator corresponding to the weaving interconnection relationship between aspect modules and class modules [84, 88].

1.5.4 Prevention Policy

The best strategy of handling conflicts is to prevent conflicts from happening. Most aspect-oriented verification approaches are based on a detection and correction strategy. Although these detection approaches are relevant for aspect-oriented software reliability,
we believe that they are time consuming and costly. It is a good thing to detect and correct system failures, but it is better to first prevent them. The fourth contribution consists of the design of the prevention policy that is used to prevent or avoid undesirable aspect interaction in aspect-oriented systems [85, 87]. This prevention mechanism will prevent most of the undesirable aspect interactions characterized by the fault classes developed in [5]. This is made up by a modular verification. This will make the verification phase much faster and cheaper.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows:

1. Chapter 2 presents the required background to understand the notions used in this thesis. It describes algebraic specification concepts and category notions.

2. Chapter 3 reviews aspect-oriented concepts, and related work on aspect-oriented modeling approaches and aspect-oriented verification approaches.

3. Chapter 4 covers the logic $L_A$.

4. Chapter 5 presents the concrete and abstract syntaxes of our approach, and also its formal semantics.

5. Chapter 6 presents the module interconnection operators and the aspect weaving algorithm.

6. Chapter 7 covers the prevention policy that we define to avoid interactions that are not desirable in aspect-oriented systems.

7. Chapter 8 presents the case study we use to illustrate our approach.

8. We conclude the dissertation in Chapter 9, and discuss potential future work.
CHAPTER 2

BACKGROUND

In this chapter, we define the required notions to understand the content of the thesis. The first section covers category theory concepts. The second section describes universal constructions in category theory. For these two sections, most of the definitions come from the books of Barr and Wells [10] and of Fiadeiro [28]. The third section presents the concepts of many-sorted set and the partial order notions. The fourth presents the concepts of algebraic specification.

2.1 Category Theory

In the book of Fiadeiro [28], category theory is presented as a toolbox similar to set theory: as a kind of mathematical lingua franca in the sense that it can be used for formalizing concepts that arise in our day-to-day activity. It provides the instruments that are more sophisticated and thus make it easier to model situations that are more complex and that involve structured objects. According to Jean-Yves Girard, category theory characterizes objects in terms of their "societal lives" (in [28]). Current software development methods, namely object-oriented ones, typically model the universe as a society of interacting objects. Agent-oriented methods are based on the same societal metaphor. Complexity does not necessarily arise from the computational or algorithmic nature of systems, but results from the fact that their behavior can only be explained as emerging from the interconnections that are established between their components (as said in [28]). We adopt this point of view, that is, category theory is a good toolbox that can be used to model software systems by focussing not only on the components of these systems, but especially on the interactions between these components.

In this section, we present the definition of few basic concepts of category theory which is a large mathematical field.

Definition 2.1. Graph

A graph $\mathcal{G}$ is a quadruple $\langle \mathcal{G}_0, \mathcal{G}_1, s, t \rangle$ where: $\mathcal{G}_0$ is a collection of nodes; $\mathcal{G}_1$ is a collection of arrows; $s: \mathcal{G}_1 \rightarrow \mathcal{G}_0$ is a function (source); $t: \mathcal{G}_1 \rightarrow \mathcal{G}_0$ is a function (target). For $f \in \mathcal{G}_1$, we write $f: X \rightarrow Y$ to indicate $s(f) = X$ and $t(f) = Y$.

Definition 2.2. Paths in a Graph
Let \( k > 0 \). In a graph \( G = \langle \mathcal{G}_0, \mathcal{G}_1, s, t \rangle \), a \textit{path} from a node \( X \) to a node \( Y \) of length \( k \) is a sequence \( (f_1, f_2, \ldots, f_k) \) of (not necessarily distinct) arrows for which:

- \( s(f_k) = X \),
- \( t(f_i) = s(f_{i-1}) \) for \( i = 2, \ldots, k \), and
- \( t(f_1) = Y \).

By convention, for each node \( X \) there is a unique path of length 0 from \( X \) to \( X \) that is denoted \( () \). It is called the \textit{empty path} at \( X \). The set of paths of length \( k \) in a graph \( G \) is denoted \( \mathcal{G}_k \). Hence:

- \( \mathcal{G}_0 \) corresponds to the collection of nodes
- \( \mathcal{G}_1 \) corresponds to the collection of arrows.
- \( \mathcal{G}_2 \) corresponds to the collection of \textit{composable pairs} of arrows.

**Definition 2.3. Category**

A \textit{category} \( \mathcal{C} \) is a graph \( \langle \mathcal{C}_0, \mathcal{C}_1, s, t \rangle \) together with two functions \( c : \mathcal{C}_2 \rightarrow \mathcal{C}_1 \) and \( u : \mathcal{C}_0 \rightarrow \mathcal{C}_1 \) with properties 1 through 4 below. The elements of \( \mathcal{C}_0 \) are called \textit{objects} and those of \( \mathcal{C}_1 \) are called \textit{morphisms} or \textit{arrows}. The function \( c \) is called \textit{composition}, and if \((g, f)\) is a composable pair, \( c(g, f) \) is written \( g \circ f \) and is called the \textit{composite} of \( g \) and \( f \). If \( X \) is an object of \( \mathcal{C} \), \( u(X) \) is denoted \( \text{Id}_X \), which is called the identity of the object \( X \).

1. \( s(g \circ f) = s(f) \) and \( t(g \circ f) = t(g) \).
2. For all \( f, g, h \in \mathcal{C}_1 \), \( (h \circ g) \circ f = h \circ (g \circ f) \) whenever either side is defined.
3. \( s(\text{Id}_X) = t(\text{Id}_X) = X \).
4. If \( f : X \rightarrow Y \), then \( f \circ \text{Id}_X = \text{Id}_Y \circ f = f \).

Sometime, we can note a category as \( \langle \mathcal{C}_0, \mathcal{C}_1, s, t, \circ \rangle \).

**Example 2.1.** Let \( \mathcal{G} \) be the class of groups and \( \text{Hom} \) the class of group homomorphisms; \( \text{dom} \) and \( \text{cod} \) are functions that associate to each group homomorphism its domain and codomain, respectively; \( \circ \) is the group homomorphisms composition function. Then, \( C = \langle \mathcal{G}, \text{Hom}, \text{dom}, \text{cod}, \circ \rangle \) is a category.
Consider $G$ and $G' \in \mathcal{G}$ and $f \in \Hom$ such that $f : G \to G'$. $f$ maps the identity element $e_G$ of $G$ to the identity element $e'_G$ of $G'$, and it also maps inverses to inverses in the sense that $f(x^{-1}) = f(x)^{-1}$.

**Example 2.2.** In the category $\Set$, objects are sets, morphisms are functions between them, composition is functional composition i.e. $g \circ f(x) = g(f(x))$ and the identity map assigns to every set the identity function on that set.

**Example 2.3.** Let $S$ be a set; $\alpha \subseteq S \times S$ a reflexive, transitive, and antisymmetric binary relation. $\langle S, \alpha \rangle$ is called an ordered set or a poset (for partially ordered set). If $\langle S, \alpha \rangle$ and $\langle S, \beta \rangle$ are posets, a function $f : S \to T$ is monotone (increasing) if whenever $x \alpha y$ in $S$, $f(x) \beta f(y)$ in $T$. Posets with monotone (increasing) functions form a category.

**Definition 2.4.** If $A$ and $B$ are objects of a category $\mathcal{C}$, then the set of all arrows of $\mathcal{C}$ that have source $A$ and target $B$ is denoted $\Hom_{\mathcal{C}}(A, B)$, or just $\Hom(A, B)$ if the category is clear from context.

**Definition 2.5.** *Functor.*

A *functor* is a structure-preserving map between categories. Let $\mathcal{C} = \langle \mathcal{C}_0, \mathcal{C}_1, s^c, t^c, \circ^c \rangle$ and $\mathcal{D} = \langle \mathcal{D}_0, \mathcal{D}_1, s^d, t^d, \circ^d \rangle$ be two categories. A functor $F : \mathcal{C} \to \mathcal{D}$ is a pair of functions $F_0 : \mathcal{C}_0 \to \mathcal{D}_0$ and $F_1 : \mathcal{C}_1 \to \mathcal{D}_1$ such that:

- $\forall X \in \mathcal{C}_0, F_0(X) \in \mathcal{D}_0$
- $\forall f : X \to Y \in \mathcal{C}_1, F_1(f) : F_0(X) \to F_0(Y) \in \mathcal{D}_1$
- $\forall X \in \mathcal{C}_0, F_1(\ID_X) = \ID_{F_0(X)}$
- $\forall f, g \in \mathcal{C}_1$ such that $g \circ^c f$ is defined, $F_1(g \circ^c f)$ is defined and $F_1(g \circ^c f) = F_1(g) \circ^d F_1(f)$.

**Remark 2.1.** In the future, $s, t, \circ$ will be used polymorphically for any category.

**Example 2.4.** For a category $\mathcal{C}$, there exists a functor $I_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$, known as the *identity functor* on $\mathcal{C}$, that maps every object and morphism in $\mathcal{C}$ to itself.

**Example 2.5.** There is functor $T : \Set \to \Set$, known as the *Cartesian square functor*, that maps every set $N$ to $N \times N$ and every function $f : N \to N'$ to $f \times f : N \times N \to N' \times N'$ where $f \times f$ is the function such that $(x, y) \mapsto (f(x), f(y))$ for all $x, y \in N$. 
**Definition 2.6. Isomorphism.**

Let $\mathcal{C}$ be an arbitrary category. A $\mathcal{C}$-morphism $f : A \to B$ is an isomorphism if there exists a morphism $f^{-1} : B \to A$ such that $f^{-1} \circ f = \text{Id}_A$ and $f \circ f^{-1} = \text{Id}_B$. The morphism $f^{-1}$ is called the inverse of $f$; and objects $A$ and $B$ are called isomorphic.

**Definition 2.7. Subcategories.**

A subcategory $\mathcal{D}$ of a category $\mathcal{C}$ is category for which:

1. All the objects of $\mathcal{D}$ are objects of $\mathcal{C}$ and all the morphisms of $\mathcal{D}$ are morphisms of $\mathcal{C}$.

2. The source and target of a morphism of $\mathcal{D}$ are the same as its source and target in $\mathcal{C}$. It follows that for any objects $A$ and $B$ of $\mathcal{D}$, $\text{Hom}_\mathcal{D}(A, B) \subseteq \text{Hom}_\mathcal{C}(A, B)$.

3. If $A$ is an object of $\mathcal{D}$ then its identity morphism $\text{Id}_A$ in $\mathcal{C}$ is in $\mathcal{D}$.

4. If $f : A \to B$ and $g : B \to C$ in $\mathcal{D}$, then the composite (in $\mathcal{C}$) $g \circ f$ is in $\mathcal{D}$ and is the composite in $\mathcal{D}$.

**Definition 2.8. Full Subcategories.**

If $\mathcal{D}$ is a subcategory of $\mathcal{C}$ and for every pair of objects $A$, $B$ of $\mathcal{D}$, $\text{Hom}_\mathcal{D}(A, B) = \text{Hom}_\mathcal{C}(A, B)$, then $\mathcal{D}$ is a full subcategory of $\mathcal{C}$.

**Example 2.6.** The category $\text{FSet}$ of finite sets and all functions between them is a full subcategory of the category $\text{Set}$ of sets and all functions; but $\text{Set}$ is not a full subcategory of category of sets and partial functions.

**Definition 2.9. Reflective Subcategories.**

Let $\mathcal{D}$ be a subcategory of a category $\mathcal{C}$. Let $C$ be an object of $\mathcal{C}$. A reflection of $\mathcal{D}$ for $C$ is a morphism $k : C \to D$ for some an object $D$ of $\mathcal{D}$ such that, for any morphism $f : C \to D'$ of $\mathcal{C}$ where $D'$ is an object of $\mathcal{D}$, there is a unique morphism $f' : D \to D'$ of $\mathcal{D}$ such that $f = f' \circ k$, i.e. the following diagram commutes:

\[
\begin{array}{ccc}
C & \xrightarrow{k} & D \\
\downarrow f & & \downarrow f' \\
D' & \xrightarrow{f''} & D'
\end{array}
\]

We say that $\mathcal{D}$ is a reflective subcategory of $\mathcal{C}$ iff every object of $\mathcal{C}$ admits a reflection of $\mathcal{D}$. 
Definition 2.10. Comma Category.

Given $F : X \to U$ and $G : Y \to U$ two arbitrary functors, the comma category $(F \downarrow G)$ has as objects the triples $(X, Y, f)$ with $X$ an object of $X$, $Y$ an object of $Y$ and $f : F(X) \to G(Y)$. A morphism from $(X, Y, f)$ to $(X', Y', f')$ is a pair $(u, v)$ with $u : X \to X'$, $v : Y \to Y'$ such that $F(v) \circ f = f' \circ F(u)$:

$$
\begin{array}{ccc}
F(X) & \xrightarrow{F(u)} & F(X') \\
\downarrow f & & \downarrow f' \\
G(Y) & \xrightarrow{G(v)} & G(Y')
\end{array}
$$

We denote $(X \downarrow G)$ the category $(1_X \downarrow G)$ with $1_X : X \to X$ the identity functor. Similarly $(F \downarrow Y)$ is defined using $1_Y : Y \to Y$.

There are two functors $P_1 : (F \downarrow G) \to X$ and $P_2 : (F \downarrow G) \to Y$ defined by $P_1((X, Y, f)) = X$, $P_1(u, v) = u$, $P_2((X, Y, f)) = Y$ and $P_2(u, v) = v$.

2.2 Universal Constructions in Category Theory

Category theory characterizes a given domain of objects by focusing on the "societal life" or interaction of these objects via morphisms. Universal constructions are standard constructions that allow expressing explicitly some relationships between objects or software components. Researchers [17, 18] showed how universal constructions can be used to express object-oriented features like encapsulation, inheritance and composition in concurrency models endowed with richer notions of state. In this section, we describe these universal constructions. As said by Fiadeiro in his book [28], intuitively, the universal constructions that are the themes of this section concern the possibility of finding objects that are able to capture the societal lives of whole collections of objects and morphisms showing how the objects relate to one another. This is because, for instance, we are interested in the study of properties of whole systems, e.g. emergent behavior, rather than isolated objects. Our goal being finding categorical concepts that can help us to handle aspect-class relationships. This section is only an entry point to the ways category theory can address this goal. It defines primitive categorical concepts that the interconnection mechanisms defined by Ehrg and Mahr [25] may use (see section 6.3 of chapter 6). This section is in fact a section of the chapter 6, but due to the fact that we have some notions which are used before this chapter, we put it here to avoid any
2.2.1 Initial and Terminal Objects

Initial and terminal objects are not constructions itself, but are distinguished objects.

**Definition 2.1. Initial and terminal objects.** Let $X$ be an object of a category $\mathcal{C} = \langle \mathcal{C}_0, \mathcal{C}_1, s, t, \circ \rangle$.

$X$ is initial if only if for each object $Y$ of $\mathcal{C}_0$, there is exactly one arrow $f : X \to Y$ of $\mathcal{C}_1$.

$X$ is terminal if only if for each object $Y$ of $\mathcal{C}_0$, there is exactly one arrow $f : Y \to X$ of $\mathcal{C}_1$.

**Example 2.7.** In the category $\text{SET}$ of the sets, the initial object is the empty set. This is because the empty set can be mapped to any other set in a unique way: through the empty function. The terminal objects are the singletons. This is because there is one, and only one way of mapping any given set to a singleton: by mapping all the elements of the source set (even if there is none...) to the element of the singleton. In Object Oriented Technology, the class that inherits from every other class can be considered as an initial object in the category whose objects are the classes; and the class from which every other class inherits can be considered as a terminal object. We take these examples from [28].

**Remark 2.2.** Initial and terminal concepts are dual notions.

2.2.2 Sum and Product

2.2.2.1 Product

Products handle the relationships from the environment towards collections of two unrelated objects.

**Definition 2.2. Cartesian product.** If $S$ and $T$ are sets, the cartesian product $S \times T$ is the set of all ordered pairs with the first coordinate in $S$ and second coordinate in $T$; in other words, $S \times T = \{(s,t)|s \in S \text{ and } t \in T\}$. The coordinates are functions $\text{proj}_1 : S \times T \to S$ and $\text{proj}_2 : S \times T \to T$ called coordinate projections or projections.

The cartesian product of two sets in category of sets is a special case of the product of two objects in an arbitrary category.
Definition 2.3. **Product.** Let $X$ and $Y$ be two objects in a category $\mathcal{C}$. A product of $X$ and $Y$ is an object $X \times Y$ together with two morphisms $\text{proj}_1 : X \times Y \to X$ and $\text{proj}_2 : X \times Y \to Y$ that satisfy the following condition: For any object $U$ and morphisms $q_1 : U \to X$ and $q_2 : U \to Y$, there is a unique morphism $q : U \to X \times Y$ such that the following diagram commutes.

\[
\begin{array}{ccc}
U & \xrightarrow{q} & X \times Y \\
\downarrow{q_1} & & \downarrow{\text{proj}_1} \\
X & \xleftarrow{i_1} & X \times Y \\
\end{array}
\]

\[
\begin{array}{ccc}
U & \xrightarrow{q} & X \times Y \\
\downarrow{q_2} & & \downarrow{\text{proj}_2} \\
Y & \xrightarrow{i_2} & X \times Y \\
\end{array}
\]

i.e., $\text{proj}_1 \circ q = q_1$ and $\text{proj}_2 \circ q = q_2$

2.2.2.2 **Sum**

The dual notion of product is sum. An object that is able to stand for the relationships that a collection $X, Y$ of objects has towards its environment is called their sum and is denoted by $X + Y$. The disjoint union of two sets in category of sets is a special case of the sum of two objects in an arbitrary category.

Definition 2.4. **Sum.** The sum, also called the coproduct, $X + Y$ of two objects in a category consists of an object called $X + Y$ together with morphisms $i_1 : X \to X + Y$ and $i_2 : Y \to X + Y$ such that for any object $V$ and morphisms $f : X \to V$ and $g : Y \to V$, there is a unique morphism $h : X + Y \to V$ for which the following diagram commutes.

\[
\begin{array}{ccc}
V & \xrightarrow{h} & X + Y \\
\downarrow{f} & & \downarrow{i_1} \\
X & \xleftarrow{i_1} & X + Y \\
\end{array}
\]

\[
\begin{array}{ccc}
V & \xrightarrow{h} & X + Y \\
\downarrow{g} & & \downarrow{i_2} \\
Y & \xrightarrow{i_2} & X + Y \\
\end{array}
\]

The morphisms $i_1$ and $i_2$ are called the canonical injections or the inclusions.

2.2.3 **Pushouts and Pullbacks**

In category theory, the means that we have for establishing interactions is through third objects that handle communication between the ones that are being interconnected via given morphisms. In the case of pushout, we are dealing with interactions at the
level of the sources, i.e. we are interested in the societal life of pairs of morphisms 
\( f : X \rightarrow Y \) and \( g : X \rightarrow Z \). In the case of pullback, the interactions are on the target 
side, i.e., \( f : X \rightarrow V \) and \( g : Y \rightarrow V \). Pushouts and pullbacks are dual notions.

### 2.2.3.1 Pushouts

**Definition 2.5.** Pushouts. In a category \( \mathcal{C} \), a pushout of a pair of morphisms \( f : X \rightarrow Y \) and \( g : X \rightarrow Z \) is an object \( W \) and a pair of morphisms \( p : Y \rightarrow W \) and \( q : Z \rightarrow W \) such that \( p \circ f = q \circ g \) and for all object \( W' \) and for all morphisms \( p' : Y \rightarrow W' \) and \( q' : Z \rightarrow W' \) such that \( p' \circ f = q' \circ g \), there is exactly one morphism \( u : W \rightarrow W' \) such that \( u \circ q = q' \) and \( u \circ p = p' \) (see the following diagram).

\[
\begin{array}{ccc}
X & \xrightarrow{f} & Y \\
\downarrow{g} & & \downarrow{p} \\
Z & \xrightarrow{q} & W
\end{array}
\]

\[\xrightarrow{q' \circ g} W'\]

In the category of sets, pushouts perform what are usually called amalgamated sums, 
i.e. they identify (join) elements as indicated by the "middle object" and corresponding 
morphisms. Pushouts are the universal constructions that allow us to join (merge) fea-
tures during multiple inheritance. The intuitive meaning of pushout in the category of 
sets is one of the union of two non-disjoint objects. \( X \) contains the elements that are 
common to the objects \( Y \) and \( Z \). The object \( W \) contains the union of \( Y \) and \( Z \) without 
repeating the elements of \( X \). A suitable pushout object needs to keep elements from both 
components as distinct as possible, while still implementing all necessary identifications, 
and without including irrelevant information.

**Example 2.8.** Consider in the category of sets,
\( X = \{a\} \)
\( Y = \{c, d\} \)
\( Z = \{b, c\} \)
\( f : X \rightarrow Y \) such that \( f(a) = d \)
\( g : X \rightarrow Z \) such that \( g(a) = b \)
The pushout of the morphisms \( f \) and \( g \) is the tuple \( \langle W, p, q \rangle \) where:
\[ W = \{e, c_1, c_2\} \]
\[
p(c) = c_2, \ p(d) = e \]
\[
q(c) = c_1, \ q(b) = e \]
\[ f(a) = d \text{ and } g(a) = b \] means that the element \( d \) of \( Y \) and the element \( b \) of \( Z \) are to be identified. Because nothing is said about \( c \), the categorical default applies: the two occurrences are to be distinguished because the fact that the same name was used is accidental.

**Definition 2.6.** Coequalizers. Let \( \mathcal{C} \) be a category and \( f, g : X \to Y \) be two parallel morphisms of \( \mathcal{C} \). A coequalizer of \( f \) and \( g \) consists of an object \( Z \) and a morphism \( e : Y \to Z \) such that:

- \( e \circ f = e \circ g \)

- For any other object \( V \) and morphism \( e' : Y \to V \) such that \( e' \circ f = e' \circ g \), there is a unique morphism \( k : Z \to V \) in \( C \) such that \( k \circ e = e' \) (see the following diagram).

\[
\begin{array}{ccc}
X & \xrightarrow{f} & Y \\
\downarrow{g} & & \downarrow{e} \\
\ & & Z \\
\end{array}
\quad\downarrow{k}
\quad e'
\quad\downarrow{V}
\]

The way to think about the coequalizer of \( f \) and \( g \) is as the quotient object made by forcing \( f \) and \( g \) to be equal.

**Proposition 2.1.** Pushouts can be obtained from sums and coequalizers.

**Proposition 2.2.** Coequalizers are a particular case of pushouts.

**Proposition 2.3.** If initial objects exist, sums can be obtained from pushouts.

The proofs of these propositions can be found in \([28]\).

### 2.2.3.2 Pullbacks

**Definition 2.7.** Pullbacks. In a category \( \mathcal{C} \), a pullback (or fibred product) of a pair of morphisms \( f : X \to Z \) and \( g : Y \to Z \) of \( \mathcal{C} \) is an object \( W \) and a pair of morphisms \( p : W \to Y \) and \( q : W \to Z \) such that \( f \circ p = g \circ q \) and for all object \( W' \) and for all morphisms \( p' : W' \to Y \) and \( q' : W' \to Z \) such that \( f \circ p' = g \circ q' \), there is exactly
one morphism \( u : W' \to W \) such that \( q \circ u = q' \) and \( p \circ u = p' \) (see the following diagram).

\[
\begin{array}{ccc}
W' & \xrightarrow{u} & W \\
\downarrow{q'} & & \downarrow{q} \\
W & \xrightarrow{p} & X \\
\downarrow{q} & & \downarrow{f} \\
Y & \xrightarrow{g} & Z \\
\end{array}
\]

Fibred products allow us to compute parallel composition with synchronization constraints \[28\].

**Definition 2.8.** Equalizers. Let \( \mathcal{C} \) be a category and \( f, g : X \to Y \) be two parallel morphisms of \( \mathcal{C} \). An equalizer of \( f \) and \( g \) consists of an object \( E \) and a morphisms \( e : E \to X \) such that:

- \( f \circ e = g \circ e \)

- For any other object \( V \) and morphism \( e' : V \to X \) such that \( f \circ e' = g \circ e' \), there is a unique morphism \( k : V \to E \) in \( \mathcal{C} \) such that \( e \circ k = e' \) (see the following diagram).

\[
\begin{array}{ccc}
V & \xrightarrow{k} & E \\
\downarrow{e'} & & \downarrow{e} \\
X & \xrightarrow{f} & Y \\
\end{array}
\]

E is called the equalizing object.

Equalizer and coequalizer are dual notions.

**Example 2.9.** In the category of sets, an equalizer \( E \) of the functions \( f : X \to Y \) and \( g : X \to Y \) is the set \( E = \{ x | f(x) = g(x) \} \). The morphism \( e : E \to X \) is the inclusion.

**Proposition 2.4.** Pullbacks can be obtained from products and equalizers.

**Proposition 2.5.** Equalizers are a particular case of pullbacks.

**Proposition 2.6.** If terminal objects exist, products can be obtained from pullbacks.

The proofs of these propositions can be found in \[28\].
2.2.4 Limits and Colimits

A limit is the categorical version of the concept of an equationally defined subset of product. For example, a circle of radius 1 is the subset of $\mathbb{R} \times \mathbb{R}$ ($\mathbb{R}$ is the set of real numbers) satisfying the equation $x^2 + y^2 = 1$. A colimit is similarly the categorical version of a quotient of a sum by an equivalence relation. Limits and colimits are dual notions. The notion of collective behavior that we wish to capture through universal constructions takes diagrams as the expression of the collection of objects and interactions that constitute what we could call a system. In fact, we tend to use diagrams to deal with complex entities for which the objects of the category provide components and the morphisms the means for interconnecting them. Hence, for instance, a typical use of diagrams is for defining configurations. The universal constructions that we address in this chapter allow us to define the semantics of such complex entities by internalizing the configuration and collapsing the structure into an object that captures the collective behavior.

An aspect of these universal constructions that is important to keep in mind is the fact that they deliver more than an object: this object comes together with morphisms that relate it to the objects out of which it was constructed. It is through these morphisms that we can understand how properties of the system (complex object) emerge from the properties of its components and the interconnections between them. Hence, the constructions are better understood in terms of structures that consist of objects together with configurations to which they relate, what are called (co)cones [28].

2.2.4.1 Colimits

The general intuition behind colimits is that they glue objects together with nothing essentially new added and nothing left over [38].

Definition 2.9. Diagram. Let $\mathcal{P}$ and $\mathcal{I}$ be graphs. A diagram in $\mathcal{I}$ of shape $\mathcal{P}$ is a homomorphism $D : \mathcal{P} \to \mathcal{I}$ of graphs. $\mathcal{P}$ is called the shape graph of the diagram $D$. $D$ is finite if $\mathcal{I}$ is finite.

Definition 2.10. Cocone. Let $\mathcal{C}$ be a category and $D : \mathcal{I} \to \mathcal{C}$ be a diagram in $\mathcal{C}$. A cocone with base $D$ is an object $Z$ of $\mathcal{C}$ together with a family $u_a : D_a \to Z$ of morphisms of $\mathcal{C}$ indexed by the objects of $\mathcal{I}$. The object $Z$ is said to be the vertex of the cocone, and the morphism $u_a$ is said to be the edge of the cocone at point $a$ for each $a$ of $\mathcal{I}$. A cocone with base $D : \mathcal{I} \to \mathcal{C}$ and vertex $Z$ is said to be commutative iff
for every arrow \( s : a \to b \) of graph \( \mathcal{G}_1 \), \( u_b \circ D_s = u_a \). The family \( u_a : D_a \to Z \) of morphisms is generally noted as \( u : D \to Z \).

The commutativity property is important because it ensures that the interconnections that are expressed in the base through the morphisms are also represented in \( z \). Hence, \( z \) is an object that is able to represent the base objects and their interactions. However, it may not do so in a "minimal" way. If a minimal representation exists, we call it a colimit of the diagram [28].

**Definition 2.11. Colimit.** Let \( \mathcal{C} \) be a category and \( D : \mathcal{G} \to \mathcal{C} \) be a diagram in \( \mathcal{C} \). A colimit of \( D \) is a commutative cocone \( u : D \to Z \) such that, for every other commutative cocone \( u' : D \to Z' \), there is a unique morphism \( f : Z \to Z' \) such that \( f \circ u = u' \), i.e., \( f \circ u_a = u'_a \) for every edge.

**Remark 2.3.** Initial objects, sums, coequalizers and pushouts are special cases of colimits.

This notion of colimit has a constructor role. They will be used to compose two or more objects (or components) interconnected and get a larger object (or component).

**Definition 2.11. Cocomplete Category.**

If all (finite) diagrams of a category have a colimit, then the category is said to be (finitely) cocomplete.

**Example 2.10.** Categories of algebraic structures are cocomplete.

**Theorem 2.1.** A category having an initial object, binary coproducts and coequalizers of parallel pairs of arrows has all finite co-limits (or is finitely cocomplete).

**Proof.** The proof of this theorem can be found in [76].

**Remark 2.4.** A category having finite coproducts and coequalizers of parallel pairs of arrows has all finite co-limits.

The proof of this remark is the same as the proof of the above theorem 2.1.

**Theorem 2.2.** Let \( \mathcal{D} \) be a full reflective subcategory of a cocomplete category \( \mathcal{C} \). Then \( \mathcal{D} \) is cocomplete.

**Proof.** The proof of this theorem can be found in [68].
**Definition 2.12.** Cocontinuous Functor.

Functors which preserve (finite) colimits are called (finitely) cocontinuous.

A colimit of a finite number of objects is called finite colimit.

**Proposition 2.1.** If $X$ and $Y$ are (finitely) cocomplete, if $G : Y \rightarrow X$ is a functor, then $(X \downarrow G)$ is also (finitely) cocomplete and the projection functors $P_1 : (X \downarrow G) \rightarrow X$ and $P_2 : (X \downarrow G) \rightarrow Y$ preserve (finite) colimits.

*Proof.* The proof of this proposition can be found in [40].

**Proposition 2.2.** Let $A : X \rightarrow U$ and $B : Y \rightarrow U$ be functors with $A$ (finitely) co-continuous. If $X$ and $Y$ are (finitely) cocomplete, then so is the comma category $(A \downarrow B)$.

*Proof.* The proof of this proposition can be found in [76].

### 2.2.4.2 Limits

**Definition 2.12.** Cone-limit. The dual notion of cocone is cone, and the dual notion of colimit is limit.

**Remark** 2.5. Terminal objects, products, equalizers and pullbacks are special cases of limits.

### 2.3 Many-Sorted Sets and Partial Order

#### 2.3.1 Many-Sorted Sets

The content of this section is taken from Sannella and Tarlecki [91]. The purpose of this section is to present the basic definitions on which the following sections rely. The basic assumption of work on algebraic specification is that a program is modeled as an algebra, that is a set of data together with a number of functions over this set. When using an algebra to model a program which manipulates several sorts of data, it is natural to partition the underlying set of values in the algebra so that there is one set of values for each sort of data. It is convenient to manipulate such a family of sets as a unit in such way that operations on this unit respect the "typing" of data values.
Definition 2.13. Many-sorted set
Let $S$ be a set (of sorts (it is like a type in a programming language)). An $S$-sorted set is an $S$-indexed family of sets $X = \{X_s\}_{s \in S}$ (or $X = \{X_s\}_{s \in S}$). Let $X = \{X_s\}_{s \in S}$ and $Y = \{Y_s\}_{s \in S}$ be $S$-sorted sets. Empty Set, Union, Intersection, Cartesian Product, Disjoint Union, Inclusion (subset), and Equality of $X$ and $Y$ are defined as follows:

$$\emptyset = \{\emptyset\}_{s \in S}$$

$$X \cup Y = \{X_s \cup Y_s\}_{s \in S}$$

$$X \cap Y = \{X_s \cap Y_s\}_{s \in S}$$

$$X \times Y = \{X_s \times Y_s\}_{s \in S}$$

$$X \uplus Y = \{X_s \uplus Y_s\}_{s \in S}$$

$X \subseteq Y$ iff $X_s \subseteq Y_s$ for all $s \in S$

$X = Y$ iff $X \subseteq Y$ and $Y \subseteq X$

Definition 2.14. Many-sorted function
An $S$-sorted function $f : X \to Y$ is an $S$-indexed family of functions $f = \{f_s : X_s \to Y_s\}_{s \in S}$; $X$ is called domain of $f$, and $Y$ is called its codomain. An $S$-sorted function $f : X \to Y$ is an identity (an inclusion, surjective, injective, bijective, ...) if for every $s \in S$, the function $f_s : X_s \to Y_s$ is an identity (an inclusion, surjective, injective, bijective, ...). The identity $S$-sorted function on $X$ will be written $\text{Id}_X : X \to X$.

If $f : X \to Y$ and $g : Y \to Z$ are $S$-sorted functions, then their composition $g \circ f : X \to Z$ is the $S$-sorted function defined by $(g \circ f)_s(x) = g_s(f_s(x))$ for $s \in S$ and $x \in X_s$.

Let $f : X \to Y$ be an $S$-sorted function and $X' \subseteq X$, $Y' \subseteq Y$ be $S$-sorted sets. The image of $X'$ under $f$ is the $S$-sorted set $f(X') = \{f_s(x) | x \in X'_s\}_{s \in S}$. $f(X') \subseteq Y$. The coimage (or inverse image) of $Y'$ under $f$ is the $S$-sorted set $f^{-1}(Y') = \{x | x \in X_s[f_s(x) \in Y'_s]\}_{s \in S}$. $f^{-1}(Y') \subseteq X$.

2.3.2 Partial Order

Definition 2.15. Binary Relation
A binary $R$ relation from a set $A$ to a set $B$ is a subset of $A \times B$. It is often convenient to write $x R y$ as shorthand for $(x, y) \in R$. If $A = B$, we will speak of a binary relation on $A$. 
**Definition 2.16. Partial Order**

A partial order \( \leq \) on a set \( A \) is a binary relation on \( A \) such that the following conditions hold:

1. \( \forall a \in A, a \leq a \) (reflexivity)
2. \( \forall a, b \in A, a \leq b \) and \( b \leq a \Rightarrow a = b \) (anti-symmetry)
3. \( \forall a, b, c \in A, a \leq b \) and \( b \leq c \Rightarrow a \leq c \) (transitivity)

We call \( \leq \) a total order on \( A \) if the following condition holds, as well:

4. \( \forall a, b \in A, a \leq b \) or \( b \leq a \)

**Definition 2.17. Partial Order Set**

A non-empty set with a partial order on it is called a partially ordered set or a poset for short. If the relation is a total order then the set is called a totally ordered set or more conveniently a chain.

### 2.4 Work of Ehrig and Mahr

#### 2.4.1 Algebraic Specification

The idea to define algebra in terms of operations and equations, was picked up by Zilles [117] to specify abstract data types like stacks, queues, and strings by algebraic equations, in an implementation or programming manner. The approach was based on specifying data types in a similar way to that used for the study of different mathematical structures, like groups, rings, etc.

Consider for example the well-known algebra, the semigroup. A semigroup \( \langle A, \star_A \rangle \) consists of a base set \( A \) and a binary operation \( \star_A : A \times A \to A \) that is associative, i.e., \( \forall a_1, a_2, a_3 \in A \), we have:

\[
(a_1 \star_A a_2) \star_A a_3 = a_1 \star_A (a_2 \star_A a_3).
\]

The set of natural numbers with the addition operation \( \langle \mathbb{N}, +_\mathbb{N} \rangle \) is a semigroup. In addition to this semantical representation of semigroup \( \langle A, \star_A \rangle \), there exist a finite syntactical representation, which is called "presentation" or specification of semigroups. Finite syntactical representations are very important in computer science [24]. Instead of a base set, a symbol "s", which is called the sort of \( A \) is only considered. The binary operation
$A \times A \rightarrow A$ is replaced by an operation symbol $o: s \times s \rightarrow s$. To express associativity in syntactical terms, one consider three variables $m_1, m_2, m_3$, elements of sort $s$ and write

$$(m_1 \circ m_2) \circ m_3 = m_1 \circ (m_2 \circ m_3)$$

Although we can write this equality syntactically, it is not an equation because the terms $(m_1 \circ m_2) \circ m_3$ and $m_1 \circ (m_2 \circ m_3)$ are not equal as syntactical units; but we obtain an equation

$$(a_1 \circ a_2) \circ a_3 = a_1 \circ (a_2 \circ a_3)$$

on the semantical level, if we assign elements $a_1, a_2, a_3 \in A$ for the variables $m_1, m_2, m_3$ respectively and replace the operation symbol $o: s \times s \rightarrow s$ by an operation $o_A: A \times A \rightarrow A$. This leads to the syntactical representation of semigroup, which is called specification of semigroup, written in figure 2.1. Likewise, the specification bool for boolean values and some operations is defined in figure 2.2: The algebra $\text{BOOL} = \langle \mathbb{B}, \text{true}, \text{false}, \neg, \land \rangle$ with $\mathbb{B} = \{\text{true}, \text{false}\}$ and the usual boolean operations $\neg$ (negation) and $\land$ (conjunction) is an algebra of the specification $\text{bool}$ (on the semantical level). The specification $\text{bool}$ (resp. the algebra $\text{BOOL}$) can be enriched by other logical operation symbols (resp. operations). So, the specifications give the representation of the syntactic domains (sorts, operation names, axioms). The part constituted by sorts and operation names is the signature of a specification, and the remaining part, i.e. the axioms, is the body of the specification. The interpretation of these syntactic domains leads to the semantic domains, which correspond to the algebras (base sets or carrier sets, operations, properties) of the specifications.

**Definition 2.18.** Classical abstract syntax of an algebraic specification [24]

An algebraic specification is a triple $\text{SPEC} = \langle S, \text{OP}, E \rangle$ where $S$ is a set of sorts, OP is a set of constant and operation symbols, and $E$ a set of equations. The pair $\Sigma = \langle S, \text{OP} \rangle$ is the signature of the specification $\text{SPEC}$.

**Remark 2.6.** Given two specifications $\text{SPEC} = \langle S, \text{OP}, E \rangle$ and $\text{SPEC}' = \langle S', \text{OP}', E' \rangle$,

<table>
<thead>
<tr>
<th>semigroup =</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sorts:</strong></td>
</tr>
<tr>
<td><strong>Opns:</strong></td>
</tr>
<tr>
<td><strong>Eqns:</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.1:** Specification of semigroup
**Figure 2.2: Specification of boolean values**

\[
\begin{align*}
\text{bool} & = \text{bool} \\
\text{Sorts: } & \text{bool} \\
\text{Opns : } & \text{TRUE, FALSE : } \to \text{bool} \\
& \text{NOT: bool } \to \text{bool} \\
& \text{AND: bool bool } \to \text{bool} \\
\text{Eqns : } & b: \text{bool} \\
& \text{NOT(TRUE) = FALSE} \\
& \text{NOT(NOT(b) = b} \\
& b \text{ AND TRUE = b} \\
& b \text{ AND FALSE = FALSE}
\end{align*}
\]

\[
\begin{align*}
\text{SPEC } \subseteq \text{SPEC}' & \iff S \subseteq S', \text{ OP } \subseteq \text{OP'} \text{ and } E \subseteq E'. \\
\text{SPEC } \cap \text{SPEC}' & = \langle S \cap S', \text{ OP } \cap \text{ OP'}, E \cap E' \rangle. \\
\text{SPEC } \cup \text{SPEC}' & = \langle S \cup S', \text{ OP } \cup \text{ OP'}, E \cup E' \rangle.
\end{align*}
\]

**Definition 2.19. Signature morphism [24]**

Given two signatures \( \Sigma = \langle S, \text{ OP} \rangle \) and \( \Sigma' = \langle S', \text{ OP'} \rangle \), a signature morphism \( h: \Sigma \rightarrow \Sigma' \) is a pair \((h_1, h_2)\) of functions \( h_1: S \rightarrow S' \) and \( h_2: \text{ OP } \rightarrow \text{ OP'} \) such that:

\[
\forall N: s_1 s_2 \ldots s_n \to s \in \text{ OP} \text{ and } n \geq 0, \text{ we have } h_2(N): h_1(s_1) h_1(s_2) \ldots h_1(s_n) \to h_1(s) \in \text{ OP'}. 
\]

**Definition 2.20. Specification morphism [24]**

Given two specifications \( \text{SPEC} = \langle S, \text{ OP }, E \rangle \) and \( \text{SPEC}' = \langle S', \text{ OP'}, E' \rangle \), a specification morphism \( h: \text{SPEC } \rightarrow \text{SPEC}' \) is a signature morphism such that the image \( h(e) \) of each equation \( e \) in \( E \) by \( h \) is a theorem in \( E' \), i.e., we can prove \( h(e) \) from \( E' \).

In our approach, each aspect or class module will contain algebraic specifications linked by specification morphisms. Relationships between system components are described by the specification morphisms.

### 2.4.2 Parameterized Specification

As saying by Parnas [70], the main goal for modular specification is to precisely specify system components without including no more information than necessary. Consider the example of the specification of the abstract data type STRING. We know that we can use strings whose elements can be naturals, reals, or letters of an alphabet, etc.

It would be better to give a unique specification of all these kinds of string instead of giving separate specifications. And these variants of string specification can be obtained
by suitable actualization of the parameter. This unique specification is called parameterized specification. Two other reasons for parameterized specification are modularity and reusability. Recall that modularity means that we want to split larger specifications into smaller parts, where each part is well-defined in itself but leaves structure and semantics of some subparts open. These subparts are the formal parameter specifications, which can be considered as "import interfaces". Reusability is important because many specifications share big parts, which should not be written from scratch again and again. They should be written only once as a parameterized specification and then reuse in different subparts of the specification.

**Remark 2.7.** If a specification $\text{SPEC}_1$ consists of a given specification $\text{SPEC}$ and additional sorts $S_1$, operations $\text{OP}_1$, and equations $E_1$, then $\text{SPEC}_1 = \text{SPEC} + (S_1, \text{OP}_1, E_1)$, where $(S_1, \text{OP}_1, E_1)$ need not to be a specification itself, because we may use sorts in $\text{OP}_1$ and operation symbols in $E_1$ which are only in $\text{SPEC}$. If $\text{SPEC} = \langle S, \text{OP}, E \rangle$, then $\text{SPEC}_1 = \langle S + S_1, \text{OP} + \text{OP}_1, E + E_1 \rangle$ where $+$ stands for disjoint union of sets.

**Definition 2.21.** Parameterized Specification.

A parameterized specification $\text{PSPEC} = \langle \text{SPEC}, \text{SPEC}_1 \rangle$ consists of a pair of specifications where $\text{SPEC} = \langle S, \text{OP}, E \rangle$ is called formal parameter specification and $\text{SPEC}_1 = \text{SPEC} + \langle S_1, \text{OP}_1, E_1 \rangle$ the target specification.

**Example 2.11.** The parameterized specification of the abstract data type STRING is given in figure 2.3, where the formal parameter specification is Data

### 2.4.3 Module Specification

Ehrig and Mahr [25] define powerful means of describing modules and a lot of classical modularity concepts such as encapsulation or genericity. Their calculus of modules is based on categorical concepts. They use the abstract data types as description formalism inside the modules. The advantages of the functional approach to programming and specification (including the avoidance of side effects, functional abstraction) advises to view modules as objects which, like functions, map "inputs" to "outputs". Programming languages with module concepts usually consider modules as their highest level program units and often distinguish different forms. Generally, they all see modules as units which consist of interface declarations expressing visibility of resources and of a body which contains the definition or realization of these resources [25]. Hence, a module specification contains an import part and an export part and must have a hiding
part (body). The reusability concept and the divide and conquer abstraction demand that modules be generic and have a distinguished parameter part. Modules form the building blocks of a modular system. Module specification is a generalization of parameterized specification [24]. Following [110], module specification offers more reusability benefit thanks to the possibility to define generic modules, and also more interaction possibility (than algebraic specification). Module specification allows to describe a component more accurately than an algebraic specification.

**Definition 2.22.** A module specification MOD is structured in four algebraic specifications, called parameter (PAR), import interface (IMP), export interface (EXP) and body (BOD), which are combined by inclusions or other specification morphisms.

\[ e : \text{PAR} \rightarrow \text{EXP} \]
\[ i : \text{PAR} \rightarrow \text{IMP} \]
\[ v : \text{EXP} \rightarrow \text{BOD} \]
\[ s : \text{IMP} \rightarrow \text{BOD} \]

such that \( v \circ e = s \circ i \). The following diagram (figure 5.9), which commutes, represents a module specification.
\[
\begin{array}{c}
\text{PAR} \xrightarrow{e} \text{EXP} \\
\downarrow i \quad \downarrow v \\
\text{IMP} \xrightarrow{s} \text{BOD}
\end{array}
\]

Figure 2.4: Module specification

- The import interface identifies those resources which are to be provided by other modules and used in the modules body for construction of resources to be exported. The explicit formulation of an import interface is useful in the stepwise development of a modular system, because it allows a top down way of construction where resources are named and used later by other modules.

- The export interface is the visible part which must be known to use this module in connection with other modules. In other words, it contains those resources which are realized by the module, but can be used by other modules. This is useful for hiding resources and serve the purpose of protection of resources.

- The parameter part is the intersection part of import and export parts in most case. It contains the possible parameters of the module. If the module is generic, it contains the parameters of the module. It aims for the reusability of the module.

- The body part contains the construction of the resources declared in export interface from those declared in import interface. This part may also contain auxiliary sorts and operations which do not belong to any other part of the module but depend on the particular choice of construction. The body part is not visible from the outside of the module.

Remark 2.8. We have in most case, \( \text{PAR} \subseteq \text{EXP} \subseteq \text{BOD} \) and \( \text{PAR} \subseteq \text{IMP} \subseteq \text{BOD} \) where \( \text{PAR} = \text{IMP} \cap \text{EXP} \). The four specification morphisms between these parts allow renaming and identification of sorts, operation symbols and formulae, and may be inclusion morphisms.

- An algebraic specification is a module specification where the import and export parts are empties.

The semantics of a module specification \( \text{MOD} \) is a functor from import specification (IMP-algebra) to export data type (EXP-algebra). A module specification can be
considered as an implementation of export by import data types, both sharing a common parameter data type (PAR-algebra).

**Example 2.12.** The module specification BOOL is given by the figure 2.5:

\[
\begin{array}{c}
\text{bool} \\
\downarrow \quad \text{Id} \\
\text{bool}
\end{array}
\begin{array}{c}
\downarrow \quad \text{Id} \\
\text{bool}
\end{array}
\]

**Figure 2.5: Module specification BOOL**

where all specification morphisms are identities.

**Example 2.13.** This example has been taken from the book [25]. Suppose a module is needed which provides a function that sorts lists of unspecified data with respect to a given but unspecified order relation on these data. Naturally such a module will export data, lists of data, and along with other operations on lists the required sorting function. It may import lists of data which it can assume to be provide by another module and it may view the data and their ordering as a parameter. A specification of such a module could schematically look as follows:

\[
\begin{align*}
\text{PAR} &= \text{Declaration of a sort data together with equality and order relation and the requirements expressing that equality is an equivalence relation and order is a total ordering on data.} \\
\text{IMP} &= \text{Extension of the parameter by declaration of lists and suitable list operations.} \\
\text{EXP} &= \text{Extension of import and parameter by declaration of a sorting function.} \\
\text{BOD} &= \text{Extension of parameter, import, and export by specification of a sorting procedure which constructs the sorting function. Depending on the chosen sorting technique, auxiliary operations may be used in the specification of the sorting procedure.}
\end{align*}
\]

where all specification morphisms are inclusions.
**Definition 2.23.** *Module morphism.*

A module morphism $m : MOD_1 \rightarrow MOD_2$ is a 4-tuple $m = (m_p, m_e, m_i, m_b)$ of specification morphisms with $m_p : PAR_1 \rightarrow PAR_2$, $m_e : EXP_1 \rightarrow EXP_2$, $m_i : IMP_1 \rightarrow IMP_2$, and $m_b : BOD_1 \rightarrow BOD_2$ such that:

$$m_e \circ e_1 = e_2 \circ m_p$$
$$m_i \circ i_1 = i_2 \circ m_p$$
$$m_b \circ v_1 = v_2 \circ m_e$$
$$m_b \circ s_1 = s_2 \circ m_i$$

The following diagram represents a module morphism. Note that this diagram commutes.

![Diagram showing module morphism](image)

**Figure 2.6: Module morphism**

### 2.5 Conclusion

We presented in this chapter the required background to understand the following chapter. We presented some basic concepts of category theory. Theorems in this part will allow to make some proofs in which category concepts are used. We describe universal constructions in category theory which may be used in chapter 6. We also presented algebraic specification technique which is one of the underline formalisms of our approach. The next chapter will cover aspect-oriented technology and the related work.
CHAPTER 3

STATE OF THE ART IN ASPECT-ORIENTED MODELING AND VERIFICATION APPROACHES

3.1 Aspect Oriented Technology

This section introduces the concepts and ideas behind aspect-oriented programming (AOP) and briefly describes basic constructs that are introduced by this paradigm.

The core of a software-based system is the subject matter of some application domain (i.e., the required items and functionalities of an application). For example, in a library, the subject matter will include Books, Periodicals, library Patrons, the Catalog of works, and so on. A system designed to manage the operations of a library must incorporate each of these subject matter items and more. Collectively, the subject matter of the system forms the core concerns of the system and constitutes the domain information that must be managed. In the implementation of the system, each core concern will typically be represented as a single abstraction with a corresponding implementation using some concrete mechanism, such as a class in an object oriented language. However, not all concerns can be represented in this discrete manner. Instead, the implementation of some concerns depends on the context provided by the behavior and concrete representation of other concerns. Such concerns are referred to as cross-cutting concerns [5].

Aspect-oriented programming is a relatively new programming paradigm that allows for the separation of concerns, especially separation of crosscutting concerns. Crosscutting concerns are concerns spread out over multiple modules, or intermixed with other concerns. To construct a system as an aspect-oriented program, one develops code for primary functionality in traditional modules and code for cross-cutting functionality in aspect modules. Following [69], AOP allows better modularity, cohesion, understandability, maintainability, and evolvability of a program or system. Yet, aspect-oriented programming brings new challenges, ideas, and concepts. Some of these concepts ¹ are:

- **Joinpoint**: a well-defined point in the execution of a program, which is used to define the dynamic structure of a crosscutting concern. Examples of join points are calling or execution of methods, access to an attribute, and initialization of an object.

¹Other definitions may exist; but we choose the standard definition in the literature.
• **Pointcut**: a set of patterns that are used to select join points. Pointcuts are defined as a predicate over the syntax-tree of the program, and define an interface that constrains which elements of the base program are exposed by the pointcut. A pattern of a pointcut is as follows:

\[
\begin{align*}
\text{pointcut} & \quad \text{pointcutName(parameters)}:\nonumber \\
& \quad \text{pointcutDefinition}; 
\end{align*}
\]

The pattern begins with the reserve word pointcut, followed by the name of the pointcut and its parameters. This defines the signature of the pointcut. The parameters indicate the origins of the joinpoints, the target, and their arguments. pointcutDefinition contains the body of the pointcut. pointcutDefinition can contain one or more joinpoints, and it is defined by primitives call, execution, get, set, handler, etc. These primitives can be combined by boolean operators in the pointcutDefinition. Wildcards can be used in the pointcutDefinition. A wildcard is a character allowing to define more extensively a method, a class, etc. For example, the pointcut pointcut methods():call(* Point.*()); designs all the method calls of the class Point for any parameter types and for any range type. Figure 3.1 defines a pointcut verifying that a caller is of type MainTest, and the called is from the class mygraphic.Point; the pointcut puts in sce the caller and in dst the called. Moreover, this pointcut verifies that the first two arguments are of the type int. One can use logical connector to combine joinpoints (e.g., or) and reserved words (e.g., this, target,...) to restrict the pointcut selection of some joinpoints.

```java
pointcut setXYValue
(MainTest sce, mygraphic.Point dst, int valX, int valY):
call(int mygraphic.Point.setXY(..)) &&
  this(sce) && target(dst) && args(valX, valY);
```

Figure 3.1: Example of a pointcut with parameters

• **Advice**: a method-like construct that contains additional behavior to be added at the matched joint point. The advice is what is woven into the join points when the pattern of a pointcut is matched. In other words, an advice is used to express the cross-cutting actions that must take place within the method body at the matched
joinpoint. Advice generally represents a fragment of control and data that must be added to the body of an existing method. There are four kinds of advices: before advice, after advice, around advice, and InsteadOf advice. A brief description of each type of advices are as follows:

- **Before advice:** executes its body before executing the body of the matched join point.
- **After advice:** executes its body after executing the body of the matched join point.
- **Around advice:** surrounds the matched join point.
- **InsteadOf advice:** replaces the execution of the matched joinpoint body.

- **Aspect:** a construct that encapsulates a cross-cutting concern. Aspects are similar to class in object-oriented programming. Besides having the properties of a class in object-oriented programming, aspects encapsulate the behavior, and state of cross-cutting concerns. Aspects include pointcuts, advices, and inter-type declarations that are used to add a public or private method, field owned by other types.

- **Aspect weaving:** the process by which behavior on aspects are merged to the original code. In other words, weaving is the process of merging the cross-cutting concerns to the core concerns of a program or system. Weaving injects the code of an aspect into joinpoints, in the syntactic structure of a primary abstraction. There are two ways in which classes and aspects can be woven: static or dynamic. Static weaving means the modification of the source code (or byte code) of a class by inserting advice at selected join points. Dynamic weaving is performed in load-time or run-time; it allows modification of a program while it is running [36].

More details will be given in the chapter 6 to show the importance of some of these concepts in the aspect-class relationship. A following figure editing system example gives an explication of the aspect concepts.

**Example 3.1. Figure Editing System.**
A simple model for figure editor system is depicted in figure 3.2. A Figure consists of a number of FigureElement, which can be either Points or Lines. The concerns of representing the display screen and the figures, points, and lines on the screen are localized by
the concrete classes Display, Figure, Point, and Line respectively. Now consider the concern of updating the display screen each time points or lines move; this concern cannot be localized in a single module in this model; its implementation cross-cuts the Point and Line modules as invocations of Display.update() in each of modifier methods of Point and Line. In this model, display updating is a cross-cutting concern. The java code of classes Point and Line depicted in figures 3.3, 3.4 shows the cross-cutting nature of Display.update() concern.

Figure 3.2: Object-oriented model for a simple figure editor system

class Point {
    private int x, y;
    ...
    public void setX(int x) {
        this.x = x;
        Display.update();
    }
    public void setY(int y) {
        this.y = y;
        Display.update();
    }
}

Figure 3.3: Java code of class Point
class Line {
    private Point $p_1$, $p_2$;
    ...
    public void setP1(Point $p_1$) {
        this.$p_1 = p_1$;
        Display.update();
    }
    public void setP2(Point $p_2$) {
        this.$p_2 = p_2$;
        Display.update();
    }
}

Figure 3.4: Java code of class Line

After separation of concerns (SoC), the classes Point and Line no longer contain the method Display.update(). Figures 3.5 and 3.6 present java code of the classes Point and Line after SoC. Figure 3.7 presents the aspect DisplayUpdating that encapsulates the cross-cutting concern Display.Updating().

class Point {
    private int $x$, $y$;
    ...
    public void setX(int $x$) {
        this.$x = x$;
    }
    public void setY(int $y$) {
        this.$y = y$;
    }
}

Figure 3.5: Java code of class Point after SoC
The best known definition of the nature of AOSD is due to Filman and Friedman [33] who characterized AOSD using the equation:

\[
\text{aspect orientation} = \text{quantification} + \text{obliviousness}.
\]

AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers. Quantification means that programs can include quantified statements (i.e., statements that apply to more than one place) of the form:

"In programs P, whenever condition C arises, perform action A".
Obliviousness implies that a program has no knowledge of which aspects modify it where or when. In the figure editor example, we can say:

\[
in the \ figure \ editing \ program, \ P, \ after \ execution \ of \ methods \ Points.setX(int), \ Points.setY(int), \ Line.setP1(Point), \ Line.setP2(Point) \ (condition \ C), \ invoke \ Display.update() \ (action \ A).\]

3.2 Comparative Study of the Main Aspect Oriented Modeling Approaches

In this section, we briefly analyze the mains aspect-oriented modeling approaches. This study allows us to highlight the need of a modular formal modeling method for aspect-oriented system development. We define criteria that we use to evaluate these modeling approaches.

3.2.1 Evaluation Criteria

We provide here some criteria, which will be used to sketch advantages and weaknesses of the main aspect-oriented design approaches found in the literature. Our criteria are partly inspired from [21], [94], and [15]. We define the new ones. We believe that these criteria are sufficient to identify the strengths and shortcomings of the existing aspect-oriented design approaches. Our collection is not exhaustive. Our aim is to briefly describe the most cited aspect-oriented design approaches in the literature. This study will help us to identify the most common limitations of these approaches as well as the major challenges in the aspect-oriented modeling and verification domain. According to us, the following criteria constitute some meaningful of a good aspect-oriented modeling approach. The first four criteria are proposed by us, while the rest are taken from other studies.

**Formalism used:** This criterion defines the formalism used, if any, that supports the approach. Through this criterion, we will see if the approach is formal, semi-formal or non-formal. The use of formal methods for software and hardware design can contribute to the reliability and robustness of a design by allowing formal verification. The value of this criterion is the name of the approach’s formalism.

**Formalism expressiveness:** Does the approach allow to easily express all the aspect concepts? Does any conflict resolution mechanism exist? The values of this criterion are the aspect-oriented concepts, like concerns separation, pointcut, advise, weaving, etc.
We believe that a good approach is an approach that clearly expresses at least the main aspect-oriented concepts. Also, we believe that an approach that is more expressive will specify a system in such a way there are less ambiguities in this specification.

**Abstraction level:** Abstraction is one of the key principles of software engineering. By leaving details aside, abstraction allows the development of tractable models, which are small enough to be verified and generic enough to be used in various language specific environments. The values of this criterion are High, Middle, and Low. If it is low, we will indicate the aspect-oriented programming language on which it is based. We agree that a more generic approach will have the high degree of abstraction.

**Verification step:** Correctness of software is critical in today’s information technology. Formal verification can provide a guarantee that a design is free of errors with regard to a specific property. Does the approach deal with the verification phase? Or, can the approach allow any verification? What is the verification approach used? Is it a formal verification? Design verification is important in software development process. The values of this criterion are yes or no. If yes, the verification approach used will be mentioned.

**Modeling tool support:** This criterion indicates if there exist tools that support the design process, or a list of guidelines that assist the way a system is designed. We know that the tool support will reduce the time of the design process of a software. We take this criterion from [94]. The values of this criterion are yes and no. If yes, the tool(s) name will be mentioned.

**Maturity:** Has the approach been applied to a real world project? The values of this criterion are yes or no. If yes, the project(s) name will be mentioned. We take this criterion from [94].

**Scalability:** Is the approach able to be applied to small as well as large projects? The values of this criterion are yes or no. This criterion comes from [21] and [15]. Scalability is good criterion that measures the software quality.

**Traceability:** Traceability allows to clearly see where an artefact comes from and where it goes to. At each level of the development process, it should be possible to identify design abstractions that represent concerns. Are the concerns identified during the requirement phase mapped onto concern abstraction at design level? Can this concern abstraction be easily mapped onto aspects in AOP language? This criterion will answer these questions in our study. We take it from [21] and [15]. Traceability is a good criterion for the maintainability, evolvability of a software.
3.2.2 Brief Study of Aspect Oriented Modeling Approaches

3.2.2.1 Selection of Aspect Oriented Modeling Approaches

Most of the aspect-oriented modeling approaches are UML-based, and for these approaches, we select the most cited in literature. We know that a few aspect-oriented modeling approaches use other formalisms in their process. In order to take advantage of these formalisms, and because the number of these approaches is not large, we systematically take any approach whose formalism-based differs from those which use UML. Our focus here is on the modeling approaches. We will do analogue study for the approaches which deal with the verification process.

3.2.2.2 Aspect Oriented Design Modeling (AODM)

Aspect-Oriented Design Modeling (AODM) [99] is a UML extension with aspect-oriented concepts. There is a relationship between AODM concepts and AspectJ concepts. AODM is not platform-independent. AODM design elements are heavily related to AspectJ language. We can therefore say that the abstraction level is low. But, it is expressive because it can represent all the major aspect concepts. An aspect is represented as a UML class with stereotype ‘< Aspect >’. Special stereotypes are introduced to model other aspectual elements. AODM approach has not been used for the modeling of a real world application. There is a wide variety of CASE tools that support AODM, since it is based on UML. We are not sure whether AODM can be used in a large real industrial project, so the scalability could be a limitation of this approach. Since there is a one-to-one mapping between AODM elements and AspectJ elements, this mapping facilitates the traceability issues.

3.2.2.3 A UML Notation for Aspect Oriented Software Design (UML-AOSD)

The JAC Design Notation [71] is designed from the JAC framework. It is an extension of UML. It is an UML notation that allows to graphically model aspect-oriented systems. The approach is platform-independent and its abstraction level is high, by using the notion of group which represents a set of objects that are not necessary instances of the class or super class. It introduces some stereotypes to model some aspect concepts. It is expressive, because it expresses the mains concepts of aspect-oriented systems. It links aspects to the base component by using the notion of pointcut relation. The paper that describes this approach does not mention any verification step and does not prove
that the approach is scalable and traceable. However, the JAC Design Notation has been already applied to real industrial projects like an online courses intranet site, a business management intranet tool. Finally, it has some modeling tools support, like ArgoUML, Rational Rose, etc.

3.2.2.4 Theme/UML

Theme approach [9] is an analysis and design approach that supports separation of concerns during analysis and design stages. It extends the UML meta-model. The Theme approach expresses concerns in conceptual and design constructs called themes. Theme is a collection of structures and behaviors that represent one feature. Cross-cutting themes are aspects. This approach uses Theme/Doc in the analysis phase to identify crosscutting concerns in requirement documents, and uses Theme/UML in the design phase to produce separate design models for each theme. It is designed to be a platform-independent AOM approach. Its composition mechanism allows resolving conflicts between themes by using special attachments or tags, and a reconciliation principle. Even thought it has been applied on various case studies, it is not clear, however, if the approach has been applied on a real world project. The composition specification is defined at a high enough level of abstraction to scale well to larger system designs. The scalability is demonstrated through a large Crystal Game application. New concerns can be added incrementally and designed separately. Theme approach maintains traceability from requirements to design, since requirements map directly to Theme/Doc views, which map directly to Theme/UML models. We didn’t find any information about modeling tools that support this approach. In Theme approach, manual verification can be done to verify that the design choices are aligned with the requirements. Concerning the expressiveness, Theme approach uses another concept to design a system. All the standard aspect concepts are not expressed in the same way like other approaches.

3.2.2.5 Aspect Oriented Software Development with Use Cases (AOSD-UC)

Aspect Oriented Software Development with Use Cases [45] is an approach that extends UML 2.0 meta-model and that uses the technology of use case to analyze and model aspect-oriented system. It allows the crosscutting concerns separation at all stage of software development life cycle. This UML extension has been influenced by the programming languages Hyper/J and AspectJ. It introduces the concept of use case slice, which encapsulates in one module the components that are specific to the realization of
a use case. Scalability of this approach is supported by this notion of use case slice. The approach is expressive in sense that it expresses almost all aspect concepts. We didn’t find any information about the verification process. The approach is generic by using a high degree of abstraction. There is one-to-one mapping between the artefact elements of analysis phase and those of design phase. The composition of aspect is differed until to the implementation phase. So, there is no composition mechanism at the modeling level. There is no validation on real industrials applications and no modeling supported tool.

3.2.2.6 Semantic-based Weaving of Scenarios (SWS)

Semantic-based Weaving of Scenarios [51] is an approach based on Message Sequence Charts (MSCs), which has been influenced by UML 2.0 sequence diagrams. But this approach only gives a behavioral aspect weaving algorithm, which is implementation platform independent. MSCs is a scenarios formalism. There are basic MSCs (bMSCs), describing simple communication patterns between system entities, and high-level MSCs (HMSCs), allowing the composition of bMSCs by the sequential, alternative, and loop operators. A HMSC is a set of bMSCs. bMSCs and HMSCs are formally defined as automata. A behavioral aspect is a pair A = (P, Ad) of bMSCs where, P is a pointcut and Ad is an advice. The approach uses the sequential composition to define the weaving process through the weaving algorithms. These algorithms are not yet implemented. The approach is expressive, but the concepts of the before, after advice are not clear. The use of HMSCs representation indicates us that the level of abstraction is high. The approach is designed as a complementary approach to others. We think that it allows formal verification because it uses a formal notation. This approach has not been applied on real industrials applications. There is no supporting tool. One can extract aspect and based models from the composed model and so, the approach facilitates traceability issues. The scalability of the approach has not been demonstrated in the paper that describes this approach.

3.2.2.7 The Motorola Weavr Approach

The Motorola Weavr Approach [19] is a UML-based approach that uses Specification Description Language (SDL) and Aspect-Oriented Modeling concepts. SDL is a specification language for the reactive and discrete systems. This approach has been developed in an industrial setting, i.e., the telecom infrastructure software industry, but it
is platform-independent. Some stereotypes, derived from the UML meta-class, are used to model aspect-oriented concepts. The Motorola Weavr Approach supports the static weaving of aspects into base models. It is expressive, but it doesn’t clearly express the different modifiers of the advice concept (connector in the motorola context). There are some conflict resolution rules. The degree of abstraction is high. This approach provides a visualization engine that allows developers to validate the join points matched by the pointcuts of an aspect, and to visualize the effects of the aspect at those locations. The composed model can be verified via simulation engine. Although the Motorola Weavr approach has been used in industrial setting, its degree of maturity is still in an initial stage. The approach is applying on a real large problem and so, it is scalable. There is a mapping between the design elements and the implementation elements.

3.2.2.8 Aspect Oriented Architecture Models (AOAM)

The Aspect-Oriented Architecture Models approach [37] is based on UML 2.0 and is platform-independent. The abstraction level is high. It uses Object Constraints Language (OCL) to specify the pre and post conditions of the operations. Concerns are modeled using class diagram templates, communication diagram templates, and sequence diagram templates. In this approach, sequence diagrams are used to realize aspect composition, but this composition is done statically. The use of so called "model composition directives" allows specifying the order in which aspect models are composed with the primary model, and also allows conflicts resolution. The approach is expressive but, it doesn’t express the notions of join point, pointcut, and advice in the same way as other approaches do. The expression of join points and pointcuts is done in composition directives. Although the authors of this approach plan to provide a model analysis, they don’t provide any detail in this paper; this model analysis could take into account some verification processes. The approach has not yet been applied on real world project. It has tools that support the modeling activities. The scalability is not supported by this approach and constitutes an open issue for the authors of this approach. The approach promotes traceability through its capability to extract aspect and primary models from the composed model.

3.2.2.9 Formalization and Verification of Aspect-UML Models with Alloy

Lightweight Formal Analysis of Aspect Oriented Models (LFAAOM) [64] is an approach, which applies a role based modeling to aspect-oriented modeling. And then,
the role-based aspect models are described in Alloy. In the context of this approach, an aspect is constituted of roles and their interaction. The aspect weaving is defined as roles merging. Weaving is essentially to make two corresponding roles merged. Weaved model is generic and can be refined using a specific platform. Although the abstraction level is high, this approach does not express clearly the aspects concepts like advice, pointcut, join point, etc. By using Alloy language, it allows formal verification and can map some requirement elements to design elements. The authors of this approach have not demonstrated any application of the approach to a real world project, and we don’t know if the approach is scalable. We don’t also identify any modeling tool support.

3.2.2.10 MATA: a Tool for Aspect Oriented Modeling Based on Graph Transformation

MATA approach [108] is an aspect-oriented modeling approach using UML and graph transformation. It considers aspect composition as a special case of model transformation. Composition of a base and aspect models is specified by a graph rule. The graph rule r: LHS → RHS defines a pattern on LHS, which represents the set of pointcuts. RHS defines the new elements to be added and specify how they should be added to the base model. MATA uses UML diagrams with some stereotypes to define the base model and aspect models. Then, it transforms these models to graphs typed and defines graph rules. It defines a mechanism for some aspect conflicts resolution. We think that this approach is expressive in sense that it expresses all aspect concepts. MATA approach is not restricted to a specific platform and the abstraction level is high. It allows some aspect interaction verification by analyzing critical pairs. MATA uses IBM’s Rational Software Modeler to model aspect and based models. MATA relies on AGG (Attributed Graph Grammar) tool to automatically apply graph rules. We can not say anything about the maturity, the scalability, and the traceability of this approach.

3.2.2.11 A Modeling Proposal for Aspect Oriented Software Architectures: PRISMA

PRISMA [73] is an approach for developing complex and large software systems, and that combines Component Based Software Development (CBSD) and Aspect Oriented Software Development (AOSD). CBSD and AOSD reduce the complexity of software development and improve its maintenance by decomposing the system into the reusable entities. PRISMA uses AOSD (Aspect Oriented Description Language) to de-
fine reusable architectural elements and aspects. This language is based on a formal object-oriented specification language, OASIS (Open and Active Specification of Information Systems). OASIS allows validation and verification processes. PRISMA's aspect-oriented architecture description language uses a graphical notation that is based on an UML profile. By defining the weaving mechanism outside the aspect, PRISMA enables aspect reusability. The weaving is defined in architectural elements (components, connectors). PRISMA approach is expressive and it is designed for complex and large systems; so it is scalable. The tool EGV-PRISMA allows to specify, in a graphical manner, PRISMA architectures and to generate automatically their PRISMA textual specification. PRISMA has also a .NET middleware and a compiler that automatically generates the source code of programs. The PRISMA approach does not depend on a specific platform and the abstraction level is high. PRISMA is mature in sense that it is already applied to real projects, for instance TeachMover robot. TeachMover robot is a tele-operation system. The reusability is an aim of the authors of this approach. We therefore deduce that this reusability promotes the traceability issue.

### 3.2.2.12 Summary of the Methods Analysis

In this section, we summarize the results of the analysis of the aspect-oriented modeling approaches we have studied above in table 3.1. For the table legend:

- yes means that the approach takes into account the criterion, no means the criterion has not been took account;
- + means the approach is expressive, +- it is less expressive, - it is not expressive;
- high means that the level abstraction of the approach is high, middle the level is middle, low the level is low;
- in means that it is in progress;
- in the "Formalism column", the name of the foundation formalism of the approach is quoted. eUML stands for extension of UML.

Most of the approaches presented above are based on UML, because aspect-oriented technology is often considered as an extension to object-oriented technology and it seems almost natural to use an extension of UML for AOM. However, because of the
<table>
<thead>
<tr>
<th>AOM Approaches</th>
<th>Formalism</th>
<th>Expressive</th>
<th>Abstraction</th>
<th>Verification</th>
<th>Tool</th>
<th>Maturity</th>
<th>Scalability</th>
<th>Traceability</th>
</tr>
</thead>
<tbody>
<tr>
<td>AODM</td>
<td>eUML</td>
<td>+</td>
<td>low</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>UML-AOSD</td>
<td>eUML</td>
<td>+</td>
<td>high</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>LFAOM</td>
<td>role-based,</td>
<td>-</td>
<td>high</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>THEME</td>
<td>eUML</td>
<td>+</td>
<td>high</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>AOSD-UC</td>
<td>eUML</td>
<td>+</td>
<td>high</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MOTOROLA</td>
<td>eUML, SDL</td>
<td>+</td>
<td>high</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>AOAM</td>
<td>UML-Pattern, OCL</td>
<td>+</td>
<td>high</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SWS</td>
<td>UML, MSC, Automata</td>
<td>+</td>
<td>high</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>MATA</td>
<td>Graph transform, UML</td>
<td>+</td>
<td>high</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>MPAOSA</td>
<td>OASIS,eUML</td>
<td>+</td>
<td>high</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Our Approach</td>
<td>algebraic specification, category theory</td>
<td>+</td>
<td>high</td>
<td>yes</td>
<td>in</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3.1: Summary results of AOM approaches analysis

The semi-formal nature of UML, one cannot formerly verify the systems built by these approaches. Also the aspect composition mechanism of these methods is not very clear.

We notice through this table that none of the approaches we have studied uses as formalisms the algebraic specification and the category theory (see the advantages of using these formalisms in sections 6.1, 2). That is one of the sources of our motivation by choosing these formalisms to model and specify aspect-oriented system. More over, algebraic specifications turned out to be adequate and flexible for dealing with structured items. A program can be modeled as algebra, i.e., a set of data together with a number of functions of this set. Category theory allows the construction of concise, comprehensible and transparent models of real-world systems. By doing this, we contribute to the resolution of the lack of formalism in aspect-oriented domain. Our approach allows the formal verification because of its mathematical background.

### 3.3 Comparative Study of the Main Aspect Oriented Verification and Validation Approaches

In this section, we briefly analyze the mains aspect-oriented verification and validation approaches. This study allows us to point out the need of a formal verification method and a consistent aspect conflict resolution mechanism for aspect-oriented system
development. Verification and Validation (VV) is the process of checking that a software system meets specifications and that it fulfils its intended purpose. It is normally part of the software testing process of a project.

3.3.1 Evaluation Criteria and Selection Criteria

This section provides criteria that we will use to evaluate the main aspect-oriented verification approaches found in the literature. But, research on testing software built via aspect-oriented programming is scarce [5]. The last six criteria (also used in [69]) have been taken from [5]. These last six criteria reflect the structural and behavioral characteristics of aspect-oriented programs. We define the new ones. We believe that these criteria are sufficient to identify the strengths and shortcomings of the existing aspect-oriented verification approaches. As for modeling approaches study, our collection is not exhaustive. Our aim is to briefly describe the most cited aspect-oriented verification approaches in literature. This study will help us to identify the most common limitations of these approaches as well as the major challenges in the aspect-oriented modeling and verification domain. By listing the main verification and testing techniques as criteria, we intend to show what are the most techniques used in aspect-oriented verification and validation approaches, and also what techniques can identify most fault types.

**Model checking:** Model checking is an automatic verification technique for finite-state systems. For a finite-state system \( S \) and a formula \( \varphi \), model checking answers whether \( S \) satisfies \( \varphi \). This criterion indicates whether the verification method used by the approach is a model checking or not. Its values are yes, no, or ? (? means that we can not conclude).

**Theorem proven:** A theorem proving is a software that proves mathematic theorems. Unlike in model checking, theorem proving solves the general validity of a formula, or a problem of whether a formula \( \varphi \) holds in all models. Theorem proving can operate on infinite state spaces but the proofs are not completely automatic. This criterion indicates whether the verification technique used by the approach is a theorem proving or not. Its values are yes, no, or ?.

**Mixing of Model checking and Theorem proven:** Due to the drawbacks of a "pure" model checking or theorem proving technique, there is a certain tentative to combine the two techniques in order to take advantage of strength of both techniques. Does the approach under consideration uses this hybrid method? The answer is yes, no, or ?.

**Static analysis:** Static code analysis is a verification technique that consists to analyze
a computer software without executing the source code of this software. The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. Does the approach under consideration uses this technique? The answer is yes, no, or ?.

**Dynamic analysis:** It is an analysis performed on executing programs. Dynamic analysis is often performed in an effort to uncover subtle defects or vulnerabilities. Does the approach under consideration uses this technique? The values of this criterion are yes, no, or ?.

**Modular verification:** Modular reasoning eliminates the need for whole program analysis before verifying a property. Is it the verification technique used in the approach? The answer of this question is yes, no, or ?.

**Aspect fault types:** An initial aspect faults model developed in [5] lists the main fault types that can arise in aspect-oriented applications due to the aspect integration into the base systems. This initial model has been refined in [7, 27, 53]. Undesirable aspect interactions are characterized, among others ², by these fault types. This faults model is summarized in six points: **Incorrect Aspect Precedence, Failure to establish expected post-conditions, Incorrect strength in pointcut patterns, Failure to preserve state invariants, Incorrect focus of control flow, Incorrect changes in control dependencies.** The description of these fault types can be found in chapter 7 of this thesis. These six fault types constitute criteria for our analysis. Can the approach detect these kinds of faults? The answer of this question is yes, no, or ?.

### 3.3.2 Brief Study of Aspect-Oriented Verification Approaches

This section summarizes our brief analysis (based on above criteria) of the main few verification approaches we found in the literature. There are few aspect-oriented verification approaches in the literature, and we take the most cited in our analysis.

#### 3.3.2.1 Data-Flow-Based Unit Testing of Aspect-Oriented Programs (DFUT)

This approach is based on a data-flow based unit testing [116]. Unit testing is to test each unit (basic component) of a program to verify that the detailed design for the unit has been correctly implemented. There are two types of unit testing, i.e., specification-based unit testing (black-box testing), and program-based unit testing (white-box test-

---

²since other sources of error may exist.
ing). Whereas specification-based testing focuses on verifying the functions and behaviors of software components according to an external view, program-based testing focuses on checking the internal logic structures and behaviors of a software component. One type of program-based testing is data flow testing which tests how values which are associated with variables can affect the execution of the program. This approach can detect the failure to preserve state invariant faults.

3.3.2.2 A State-Based Approach to Testing Aspect-Oriented Programs (STAOP)

An aspect can change the state model of the core concerns (classes) through introduction and advices. An aspect can introduce new states into the system and can cause the classes to violate their state invariants. STAOP [114] is a state-based approach to testing aspect-oriented programs. It is an extension of FREE state model to aspectual state model (ASM). FREE (Flattened Regular Expression) state model was designed to provide a testable model for class behavior. ASM is developed to provide a testable model of class behaviors along with additional advices defined in aspects, which are dynamically attached when specific join points are reached. ASM is then transformed to a transition tree, which represents transitions for events. This transition tree implies a test suite for adequately testing object behavior and interaction between classes and aspects. STAOP uses the N+ testing strategy, a strategy that allows exercising a program through all the paths of the conditional transition tree. This testing strategy can reveal all state transition faults, corrupt states, sneak paths, and aspect faults such that incorrect strength in point-cut patterns, and failure to preserve state invariants (according to [114]).

3.3.2.3 Un Cadre Formel pour le Développement Orienté Aspect: Modélisation et Vérification des Interactions dues aux Aspects (ALLOYM)

Mostefaoiu used Alloy [44] in her thesis [62] to verify and detect interactions errors due to the aspects weaving into the base system. Alloy is a structural modeling language based on the first order logic. Alloy allows automatic verification based on constraints resolution and not on state enumeration. Alloy verification, by its analyzer, allows assertion validation by generating counterexamples. Her approach consists to define into Alloy language a generic pattern for each kind of property to be verified. To verify a property, an instance of its corresponding pattern is executed with the Alloy analyzer. If a counterexample is generated, then there is an interaction problem in the system. The verification of local properties allows to detect "Failure to establish expected post-
conditions" fault type, and the global properties one can detect "Failure to preserve state invariants" fault type. Although this verification is consistence and does not encounter the state space explosion problem, it is incomplete because it considers only small models.

3.3.2.4 Trace Analysis for Aspect Application (TAAA)

Using program traces, changed behavior becomes observable as here analysis of the internal program flow is possible. So this can be an adequate mean to analyze the effects of advice application to a given system. This approach [100] relies on comparison of two traces for a single test: one trace is generated with the (base) system without applied aspects and the other is generated by the woven program. By identifying patterns of differences, one can observe changed behavior. Even if we cannot explicitly infer, we think that the proposed approach can probably detect the fault types "Failure to establish expected post-conditions" and "Incorrect changes in control dependencies". As said by the authors themselves, the proposed trace analysis does not show when an aspect has accidentally not been applied to some parts of the system. More, total coverage is needed to be able to get more reliable results.

3.3.2.5 Model Checking Aspect Oriented Design Specification (MCAODS)

MCAODS [115] is an approach to model-checking state-based specification of aspect-oriented design. An aspect-oriented state model consists of class models, aspect models, and aspect precedence. A (class, aspect) state model consists of states, events, and transitions. Aspect precedence is specified by a partial order relation on the given set of aspect models. To verify an aspect-oriented state model, the authors of this approach first weave aspect models into their base class models. This results in woven state models. Then they transform the woven models and the models of those classes not modified by the aspects into respective FSP (Finite State Processes) processes. FSP are textual notations to describe system models. They also express the desired system properties as property processes. They finally apply the LTSA model checker (in [115]) to verify the generated FSP processes against the desired system properties. If a property is violated, LTSA generates a counterexample. This approach can detects "Incorrect aspect precedence" and "Failure to preserve state invariants". It has model checking advantages and limitations.
3.3.2.6  Aspect-Oriented Programming with Model Checking (AOPMC)

In this paper [103], an AOP-based model checking framework is proposed in order to use model checkers efficiently. It used the woven code as a model input. Using this checking framework, properties to be checked that crosscut over classes can be described as an aspect and separated from a program body. Design by contract style is used to build this aspect. The assertions are based on some kinds of joinpoint in AspectJ such as call and set joinpoints. A checking property is expressed as an aspect that describes pre- and post-conditions using joinpoints. This checking framework is usually used with a tool (such as Jpf to detect deadlocks for example) to cover all testing features. But, what happen if a tool for a specific testing feature does not exist? This approach can detect the failure to preserve state invariant and failure to establish expected post-conditions faults. This approach uses a detection strategy while our approach is based on a prevention and detection strategies. Authors of this paper stated that there are problems that should be resolved in order to practice their approach.

3.3.2.7  Modular Generic Verification of LTL Properties for Aspects (MGVPA)

MGVPA approach [41] uses a modular technique in which the aspect can be considered separately from the base program. Both base programs and aspect are defined as nondeterministic finite state machines in which particular computations are realized as infinite sequences of states within the machine. The specification of an aspect consists of formulae expressing assumptions about any base machine to which the aspect can be woven, and formulae expressing the desired properties to be satisfied by any augmented machine (woven machine). The formulae are expressed in linear temporal logic (LTL). Then, when a particular base program is to be woven with the aspect, it is sufficient to establish that the base state machine satisfies the assumptions. Thus the entire augmented program never has to be model checked, achieving modularity and genericity in the proof. However, we are wondering about the obliviousness of AOP with regard to this approach. More, the authors of this paper do not explain what kind of properties can be verified by their approach. Also, we believe that LTL is not too expressive to express all types of aspect interactions. But, we infer through the description of this approach given in [41] that "Failure to establish expected post-conditions" and "Failure to preserve state invariants" can be detected. MGVPA addresses only a single aspect and not multiple aspects at a specific joinpoint.
3.3.2.8 Testing Aspect-Oriented Programs with UML Design Models (TAOP-UDM)

This approach [113] presents a model-based approach to testing whether or not an aspect-oriented program conforms to its expected crosscutting behavior specified by aspect-oriented UML design models. The testing process consists of the following steps: (1) building an aspect-oriented UML model; (2) generating test cases from the aspect-oriented model; (3) executing tests by feeding test inputs to the aspect-oriented program and the test harness; (4) determining whether a test case passes or fails by comparing the actual result of the test execution with the expected result of that test. Each aspect-oriented model consists of class diagrams, aspect diagrams, and sequence diagrams. For a method under test, the sequence diagrams of the advice on the method are woven into the method’s sequence diagram. Based on the woven sequence diagram and class/aspect diagrams, an AOF (Aspect-Object Flow) tree is then generated by applying coverage criteria such as condition coverage, polymorphic coverage, and loop coverage to woven sequence diagrams. In the AOF tree, each path from the root to a leaf is an abstract message sequence, indicating a template of test cases. A concrete test case is obtained by creating objects that satisfy the collective constraints in the template. Their empirical study shows that the model-based testing approach is capable of revealing several types of aspect-specific faults, incorrect (weaker or stronger) pointcut strengths, and incorrect aspect precedence.

3.3.2.9 Summary of the Methods Analysis

In this section, we summarize the results of the analysis of the aspect-oriented verification and validation approaches we have above studied in table 3.II. For the table legend:

- yes means that the approach uses this technique or can detect this fault type;
- no means it does not use this technique or can not detect this fault type;
- ? means that we can not conclude;
- a means that our approach can avoid this fault type;
- in means that it is in progress.
<table>
<thead>
<tr>
<th>AOVV Approaches</th>
<th>Model checking</th>
<th>theorem proven</th>
<th>Mising of MC and TP</th>
<th>Static analysis</th>
<th>Dynamic analysis</th>
<th>Modular verification</th>
<th>Incorrect strength in point-cut patterns</th>
<th>Incorrect aspect precedence</th>
<th>Failure to establish expected post-conditions</th>
<th>Failure to preserve state invariants</th>
<th>Incorrect focus of control flow</th>
<th>Incorrect changes in control dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFUT</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>?</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>STAOP</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>ALLOYM</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>TAAA</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>?</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>MCAOIDS</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>?</td>
<td>yes</td>
<td>?</td>
<td>yes</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>AOPMC</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>?</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>MGVPMA</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>?</td>
<td>yes</td>
<td>?</td>
<td>yes</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>TAOPUDM</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><strong>Our Approach</strong></td>
<td><strong>in</strong></td>
<td><strong>in</strong></td>
<td><strong>in</strong></td>
<td><strong>no</strong></td>
<td><strong>no</strong></td>
<td><strong>yes</strong></td>
<td><strong>a</strong></td>
<td><strong>a</strong></td>
<td><strong>a</strong></td>
<td><strong>a</strong></td>
<td><strong>a</strong></td>
<td><strong>in</strong></td>
</tr>
</tbody>
</table>

Table 3.2: Summary results of aspect-oriented VV approaches analysis

There are two mains approaches which use model checking technique. According to our present knowledge, there is no approach using theorem proving. MGVPMA approach [41] is only one approach that uses a modular technique. It is a little similar to our approach. However, our approach is more general. MGVPMA does not use the structure of the system to do the verification task and addresses only one aspect at a joinpoint. All these approaches focus on one or two specific undesirable aspect interactions, while our goal is to provide a framework to many types of undesirable aspect interactions. The approach TAOPUDM described in the paper [113] is one acting at the modeling phase and can detect incorrect (weaker or stronger) pointcut strengths, and incorrect aspect precedence. These verification approaches are based on conflict detection, contrary to our approach which is based firstly on a prevention strategy and then on a correction strategy if it is necessary.

### 3.4 Thesis of Wiels

We decided to use or to extend Wiels [110] work because of the modularity benefits offered by her approach at the specification and verification levels. In this section, we summarize this approach. This work had been motivated by the need of modular system development framework to deal with the complexity of the software systems. At the implementation level, programming languages like object-oriented languages have emerged to manage this complexity in modular way. However, in the formal methods
area, the other phases of the software development process, such as specification, design, and validation phases, lack of the similar concepts used in the implementation step.

The proposed approach combines algebraic modules and a temporal logic of action. Several authors such as Fiadeiro and Maibaum [31] have noticed that a combination of temporal logic and algebraic structure based on categories is interesting because temporal logic is very expressive and well-suited to the specification of reactive systems while categories allow to structure the logical descriptions and provide composition laws. Wiels used a logic she had designed in [109]. This temporal logic of action is a combination of linear temporal logic and dynamic logic. Wiels approach is expressive and modular. Expressiveness, given by the use of logic, is necessary to specify complex systems. She adapted the calculus of modules proposed by Ehrig and Mahr [25], by using her logic to describe the behavior of the module parts. She used construction operators defined by Ehrig and Mahr to compose modules.

Wiels expresses equations or formulae in this logic and a specification is defined as a tuple (Sorts, Operations, Attributes, Actions, Logic Formulae). In Wiels approach, there are three levels of system description:

- a system is described by modules that are interconnected by morphisms and on which composition operations can be performed;
- a module is composed of four specifications linked by specification morphisms;
- each specification is a logical theory (a signature that gives the vocabulary (attributes and methods) and a set of formulae to describe the behavior).

A category is built. The objects of this category are the module specifications and its morphisms are the module morphisms. The module representing the global system under development is obtained by taking the colimit of the diagram constituted by the module specifications and module morphisms.

This approach, by taking modularity advantage offered by category theory, allows modular verification. It is supported by a tool that implements all the categorical concepts and operations needed for the definition of modules and thus automatically builds the specification of systems from the specifications of their components. The kernel of this tool is based on the ideas of Rydeheard and Burstall [76] and is realized by Jacques Sauloy [92], [93]. This kernel defines all the necessary useful notions of category theory. Above this kernel, the calculus of modules is implemented; a part of this level was realized by Jacques Sauloy. A third level provides user services like a parser and a
graphical interface. Currently, this tool is in standard ML (SML). SML is a functional language. We need to rebuild it in an object oriented language such as OCAML, by integrating aspect concepts or we can also use an aspect oriented language such as As-pectual CAML [56]. We think that the development of this tool will take a lot of time. Finally, she applied this approach to some case studies such as telecommunication system: "specification of a Notification Selection and Dispatching Function (NSDF)" and avionic system: "specification of an Electrical Flight Control System (EFCS)".

Figure 3.8 gives a parallel of our approach with Wiels’. Wiels approach is motivated by the need to import the modularity concepts of object-oriented programming (OOP) into formal specification and verification phases. Her goal is to handle the complexity of system development at the formal specification and verification levels. As stated in section 1.3, our problematic are: (1) the need of aspect-oriented formal specification and verification methods and (2) the aspect interaction problems. Our goal is to import AOP modularity concepts into formal specification and verification phases, improve trace-ability, and resolve aspect interaction problems, by designing a formal framework for the specification and the verification of aspect-oriented system. AOP is an extension of OOP. Our approach is an extension of Wiels approach which uses Ehrig and Mahr framework. By extending Wiels approach, we extend among others, the module calculus of Ehrig and Mahr. We use a combination of temporal, dynamic, and deontic logics while she uses temporal and dynamic logics. The main differences between our approach and Wiels’ one are: our approach is intended for aspect-oriented systems while her approach is for object-oriented systems; we use a combination of temporal, dynamic, and deontic logics while she uses temporal and dynamic logics.

3.5 Conclusion

In this chapter, we introduced aspect-oriented technology, and we briefly analyzed the mains aspect-oriented modeling and verification approaches. This study allows us to highlight the need of a modular formal modeling method for aspect-oriented system development. Most of the aspect modeling approaches presented above are based on UML. Because of the semi-formal nature of UML, one cannot formerly verify the systems built by these approaches. Also the aspect composition mechanism of these methods is not very clear. This study also allows us to point out the need of a formal verification method and a consistent aspect conflict resolution mechanism for aspect-oriented system development. All the verification approaches focus on one or two specific undesirable aspect
interactions, while our goal is to provide a framework to many types of undesirable aspect interactions. These verification approaches are based on conflict detection, contrary to our approach which is based firstly on a prevention strategy and then on a correction strategy if it is necessary. There are formal works such as the one in [97, 106] which act at the implementation level but we are dealing with the specification and verification phases of the software development process. Moreover, our framework is intend to be programming language independent. Finally, we described Wiels’ approach that we extend, by showing the main differences between our approach and hers.
CHAPTER 4

LOGIC $\mathcal{L}_A$

4.1 Introduction

We want a logic, expressive enough, which can allow to describe the behavior of class and aspect components, and also the societal life of these components. This societal life will be expressed in the form of prescription properties that prevent undesirable interactions between these components. Also, we want our approach to be able to specify properties of many kind of systems. Thus, we include in our logic $\mathcal{L}_A$ linear temporal logic (LTL) [55], (first-order) dynamic logic (FDL) [43], and deontic logic (DL) [112]. $\mathcal{L}_A$ contains other modalities corresponding to the aspect modifiers Before, After, Around, and InsteadOf. LTL is a simple logic that has been used in many specification works of concurrent and reactive systems [55]. LTL allows us to reason on the time and about the future. It models time as a sequence of states, starting at 0 and extending infinitely into the future. FDL is type of modal logic, which is intended to reason about computer programs and propositions by integrating the notion of actions (which are computer programs or programs for short). For instance, $[a]p$ means that after each execution of the action $a$, the property $p$ must hold. It is worth noticing that LTL cannot reason explicitly about actions contrary to FDL. DL is a sub-domain of the modal logic, whose goal is to describe the normative systems. Normative systems model a set of interacting objects or agents whose behavior is governed by social norms. Deontic logic is the logic that deals with actual as well as ideal behavior of systems. Wiels [110] used a combination of LTL and dynamic logic in her approach because this combination offers means to reason over time and also actions. Brunel [14] used a combination of temporal logic and deontic logic to formally specify security policy for computer systems. For example, he specifies resource access control with his logic. Prisacariu and Schneidder [75] use a combination of temporal logic, DL, and propositional dynamic logic to represent and reason about contracts. Contracts are used to regulate the interaction and exchanges between the parties involved (being that services, components, or agents) in service oriented architectures, component based systems, and agent societies. Deontic logic can be used in situations where the established rules may be violated. Deontic logic allows to reason about situations in which not all that we want is true.

$\mathcal{L}_A$ will allow us to describe not only the behavior of aspect and class components,
but especially to regulate the societal life of these components. Others have shown that such a combination (LTL + Dynamic Logic + Deontic Logic) is adequate for systems specification [42, 48], because it can distinguish between description and prescription of behavior. Action prescriptions are means to convey when actions may or must occur, through the deontic concepts of obligatory and permissible actions. The description is achieved by the traditional pre and post condition style description of actions. This then allows us to state when actions may and must happen as opposed to just describing the effects of such actions. We attach an operational meaning to the notion of permission and obligation in terms of which we can speak about the normative behaviors of an object or an aspect instance: those where actions occur only when permitted and where obligatory actions are performed otherwise there will be a punishment or a reparation. These normative behaviors characterize the safety and the liveness properties of an object or an aspect. Our logic will allow to partition the set of object and aspect interactions into two kind of interactions: normative interactions and non-normative interactions. Intuitively, a normative interaction is one which is behaviorally desirable and has been deontically sanctioned, whereas a non-normative interaction is behaviorally undesirable and has been deontically prohibited [48]. In our approach, we will specify both normative and non-normative interactions. We adopt the -do approach where deontic operators are applied to actions. It is argued in the deontic community that this would avoid several of the present paradoxes. The ought-to-do approach has been advocated by von Wright [105] which argued that deontic logic would benefit from a "foundation of actions", and many of the philosophical paradoxes of standard deontic logic (SDL) would be eliminated. In ought-to-be approach, deontic operators are applied to state-of-affairs. For example, "Peter ought to take his medicine" is an ought-to-do sentence while "Peter takes his medicine" is an ought-to-be sentence.

The behavior of the desired aspect interactions is violable. Hence, contrary-to-duty obligations (CTD) and contrary-to-prohibitions (CTP) have to be considered in our logic. CTDs are statements that represent the fact that the obligation might not be respected while CTPs are similar to deal with prohibitions that might be violated. Both constructions specify the obligation/prohibition to be fulfilled and the reparation/punishment to be applied in case of violation [75]. CTDs statements are usually used in fault-tolerance, where an obligation to perform an action arises after another obligation was not fulfilled. CTDs and CTPs are useful to write our prevention policy. In fact, the reduction of the deontic operators to the dynamic operators takes into account CTDs and CTPs principles in our logic.
A model of \( L_A \) represents a set of traces of a possible execution of a system. Each instant of this model describes a system state. A state is an assignment of values to (class or aspect instance) attributes at this instant (for example, the attributes of a person can be his name, age, address, etc.). Transitions between states are labeled with atomic actions. We have been inspired by the work of [95] to define \( L_A \). The difference between \( L_A \) and their logic is that we have integrated deontic and aspect concepts while their logic have not.

4.2 Syntax of \( L_A \)

Before defining the syntax of \( L_A \), we introduce some notations: \( \Phi_0 \) a set of atomic propositions denoted by \( p, q, \ldots \), \( \Phi \) a set of propositions or formulae, \( \Phi_0 \subseteq \Phi \). \( \mathcal{A} \) a set of atomic actions denoted by \( a, b, \ldots \) \( \mathcal{A} \) a set of actions denoted by \( \alpha, \beta, \ldots \) \( \mathcal{A} \subseteq \mathcal{A} \).

**Definition 4.1. Syntax of pure formulae**

The syntax of a formula is given in Backus Naur form (BNF) form by:
\[
\phi ::= \bot \mid T \mid p \mid \neg \phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \rightarrow \phi \mid \phi \leftrightarrow \phi \mid \forall x \phi \mid \exists x \phi \mid X \phi \mid G \phi \mid F \phi \mid \phi \cup \phi \\
\mid [\alpha] \phi \mid < \alpha > \phi \mid \phi \rightarrow \alpha/\beta \mid (B_\alpha) \alpha \mid (A_\alpha) \alpha \mid (A, \phi) \alpha \mid (I_\alpha) \alpha
\]
where \( p \in \Phi_0 \), \( \alpha, \beta \in \mathcal{A} \), \( x \) is a variable. A pure formula or formula for short is simply a proposition.

**Definition 4.2. Syntax of action formulae**

The syntax of an action formula is given in BNF form by:
\[
\alpha ::= \top \mid \bot \mid \phi \mid a \mid V \mid \alpha \wedge \alpha \mid \alpha \wedge \alpha \mid \alpha \rightarrow \alpha \mid [\alpha] \alpha \mid < \alpha > \\
\alpha \mid I \alpha \mid \alpha \mid P \alpha \mid \neg \alpha \mid O_k \alpha , \text{ where } a \in \mathcal{A} \}, \phi \text{ is a proposition and } k \text{ is an positive integer.}
\]

We give an informal meaning of these formulae and action formulae.

- \( X \phi \) is a formula meaning that \( \phi \) will hold in the next state.

- \( G \phi \) is a formula meaning that \( \phi \) holds in the current state and will always hold in the future state.

- \( F \phi \) is a formula meaning that there exist a future state in which \( \phi \) holds.

- \( \phi \cup \psi \) is a formula meaning that \( \phi \) holds until \( \psi \) holds.
• $[\alpha] \beta$ is an action which means that immediately after each execution of the action $\alpha$, the action $\beta$ must be executed (necessity).

• $< \alpha > \beta$ is an action which means that it is possible to execute the action $\alpha$ and reach a state in which the action $\beta$ has to be executed (possibility).

• $\alpha; \beta$ is an action which means that the action $\alpha$ is executed first followed by the execution of $\beta$ (sequential composition).

• $\alpha \lor \beta$ is an action which means that the action $\alpha$ or $\beta$ is executed non-deterministically (non-deterministic choice).

• $\alpha \land \beta$ is an action which means that the actions $\alpha$ and $\beta$ are executed simultaneously (parallel composition).

• $\alpha^*$ is an action which means that the action $\alpha$ is executed sequentially $n$ times ($n \geq 0$) (non-deterministic iteration).

• $\neg \alpha$ is an action indicating that the action $\alpha$ is not executed.

• $I \alpha$ is an action which means that it is forbidden to execute the action $\alpha$, otherwise there will be a punishment represented by a constant action $V$.

• $V$ is an action in our logic, contrary to the case of the constant of Anderson [6].

• $O \alpha$ is an action which means that it is obligatory to execute the action $\alpha$, otherwise there will be a reparation represented by a constant action $V$.

• $O_k \alpha$ is an action which means that it is obligatory to execute the action $\alpha$ before the date $k$, $k$ being a positive integer. We take this modality from [14].

• $P \alpha$ is an action which means that it is permissable to execute action $\alpha$.

• $[\alpha] \phi$ is a formula meaning that the formula $\phi$ holds after all execution of the action $\alpha$ (necessity).

• $< \alpha > \phi$ is a formula meaning that it is possible to execute the action $\alpha$ and reach a state in which the formula $\phi$ holds (possibility).

• $\phi \rightarrow \alpha/\beta$ is a formula which means that if $\phi$ holds in the current state, then the action $\alpha$ is executed otherwise the action $\beta$ is executed (conditional).
• \( \phi \) is an action which tests if the formula \( \phi \) is true. If \( \phi \) is true, then proceed otherwise fail (test). Dynamic logic associates to every proposition \( \phi \) an action \( \phi \) called a test. If \( \phi \) holds, the test \( \phi \) acts as a skip, changing nothing while allowing the action to move on. If \( \phi \) is false, \( \phi \) acts as an "abort".

It is worth noting that \( \phi \rightarrow \alpha/\beta \) is not the same as the formula \( \phi \rightarrow \alpha \) because the later is some what problematic, since we don’t know its status when \( \phi \) does not hold. In the following four formulae, \((B_c\phi)\alpha\), \((A_f\phi)\alpha\), \((A_r\phi)\alpha_1\alpha_2\) and \((I_o)\alpha\), \( \phi \) is an atomic proposition meaning that a joinpoint \( jp \) is reached.

• \((B_c\phi)\alpha\) is a formula which means that just before each state \( s \) where the formula \( \phi \) is true, the action \( \alpha \) is obliged to be executed (before). We read \((B_c\phi)\alpha\): before \( \phi \alpha \).

• \((A_f\phi)\alpha\) is a formula which means that immediately after each state \( s \) where the formula \( \phi \) is true, the action \( \alpha \) is obliged to be executed (after). We read \((A_f\phi)\alpha\): after \( \phi \alpha \).

• \((A_r\phi)\alpha_1\alpha_2\) is a formula which means that in each state \( s \) where the formula \( \phi \) is true, \( \alpha_1 \) is obliged to be executed immediately before and \( \alpha_2 \) immediately after the action that should be executed in \( s \) (Around). We read \((A_r\phi)\alpha\): around \( \phi \alpha \).

• \((I_o)\alpha\) is a formula which means that in each state \( s \) where the formula \( \phi \) is true, the action \( \alpha \) is obliged to be executed instead of the action (encapsulated in the body of the joinpoint \( jp \)) that should be executed in \( s \) (InsteadOf). We read \((I_o)\alpha\): instead of \( \phi \alpha \).

Remark 4.1. \( (A_r\phi)\alpha_1\alpha_2 = ((B_c\phi)\alpha_1) \land ((A_f\phi)\alpha_2) \)

Examples of \( L_a \) formulae are: \([\alpha]G(\phi \land \psi), O(\phi?), (B_c\phi)\alpha \lor \beta \).

4.3 Semantics of \( L_A \)

Programs or actions will be interpreted by binary relations defined on states of a transition system, and formulae by sets of states. We interpret the formulae and the actions on a structure \( M = \langle W, r, l, \beta, \gamma, pr \rangle \), where \( W \) is a set of states:

\[
  r : \mathcal{A}_I \rightarrow 2^{W \times W} \\
  a \mapsto r(a)
\]
is a function which associates to each atomic action \(a\), a binary relation \(r(a)\) s.t. \(\forall x, y \in W, x \ r(a) \ y\) iff there exist an execution of \(a\) from the state \(x\) to the state \(y\):

\[
\begin{align*}
l : & \quad \Phi_0 \rightarrow 2^W \\
p & \mapsto l(p)
\end{align*}
\]

is a function which associates to each atomic proposition \(p\) a subset \(l(p) \subseteq W\) s.t. \(\forall x \in W, x \in l(p)\) iff \(p\) holds in \(x\); functions \(r\) and \(l\) are inductively extended to the complex actions and formulae. \(l(\varphi) \subseteq W\) for a formula \(\varphi\) and \(r(\alpha) \subseteq W \times W\) for an action \(\alpha\). Intuitively, we can think of the set \(l(\varphi)\) as the set of states satisfying \(\varphi\) in the model \(M\) and we can think of the binary relation \(r(\alpha)\) as the set of input/output pairs of states of the action \(\alpha\). As said Meyer [58], one has to imagine that one is in a state (world) \(x\), in which certain assertions hold. Then by doing an elementary action a one moves to a next state \(x'\). We assume actions to terminate after a finite amount of time. In this state \(x'\) other assertions may hold than in \(x\), since \(a\) might have changed something.

The semantics assumes given a set \(J\) (of joinpoints) and two functions defined on \(J\); the function \(f_j : J \longrightarrow \mathcal{O}f\) that associates to a joinpoint \(jp\), the action \(f_j(jp)\) encapsulated in the body of \(jp\); the function \(pr : J \longrightarrow \Phi_0\) that associates to a joinpoint \(jp\), the (atomic) proposition "the joinpoint \(jp\) is reached" such as:

\[
\begin{align*}
pr(jp_1||jp_2) & = pr(jp_1) \lor pr(jp_2) \\
pr(jp_1\&\&jp_2) & = pr(jp_1) \land pr(jp_2)
\end{align*}
\]

If the joinpoint \(jp_1\) is reached before the joinpoint \(jp_2\), we note \(pr(jp_1) \preceq pr(jp_2)\).

\(\langle W, r, l \rangle\) is a Kripke structure used in [8, 43] to interpret formulae and actions. As in [43], we take an operational view of program semantics: programs change the values of variables by sequences of simple assignments \(x := t\) or other assignments, and flow of control is determined by the truth values of tests performed at various times during the computation.

**Definition 4.3.** Let \(\langle X, \leq_o \rangle\) be a partial order set. If \(x \leq_o y\) and \(x \neq y\), then we write \(x <_o y\) and we say that \(x\) is a predecessor of \(y\). \(x \leq_o y\) means that \(x <_o y\) or \(x = y\). If \(x \leq_o y\) and if \(y \leq_o z\), we say that \(y\) is between \(x\) and \(z\). If \(x <_o y\) and if \(y <_o z\), we say that \(y\) is strictly between \(x\) and \(z\). If \(x <_o y\) and if there is no element strictly between \(x\) and \(y\), we say that \(x\) is an immediate predecessor of \(y\), or \(y\) is the immediate successor of \(x\) and we write \(x <^i_o y\).

We assume that there is an order \(\leq_o\) over \(W\) in the sense of definition 4.3.
Definition 4.4. A path in $M$ is a finite sequence of states $\pi = s_0s_1...s_n$ of $W$ (n is natural number) such that there exists a sequence $a_i$ of actions such that $\forall 0 \leq i < n, s_i r(a_i) s_{i+1}$.

The path $\pi = s_0s_1...$ represents a possible future of a system: first it is in state $s_0$, then it is in state $s_1$, and so on. We write $\pi^i$ for the suffix starting at $s_i$, e.g., $\pi^4 = s_4s_5...$. Let $\pi = s_0s_1...$ be an execution path of $M = \langle W, r, l, fj, pr \rangle$; $\phi$ and $\psi$ two formulae, $\alpha, \beta$ two actions. $V$ is a constant action representing a punishment or a reparation (that indicates a rule or a norm is violated). By definition, $\bot = \text{abort}$ (which means fail or impossible action since it leads nowhere) and $\top = \text{skip}$ (which means proceed or continue, or don’t care what happens). $r(\bot)$ is the empty relation and $r(\top)$ is the identity relation. We do not require $\neg\alpha$ to be a set-complement of $\alpha$ in the sense that $\alpha \cup \neg\alpha = \mathcal{A}$. We do not impose this requirement because it implies that $\alpha = \mathcal{A} \setminus \{\neg\alpha\}$. We adopt the same meaning for an action negation as in [58]. We consider $\neg\alpha$ such that the following properties hold:

\[
\neg \neg\alpha = \alpha \\
\neg(\alpha; \beta) = \neg\alpha \lor (\alpha; \neg\beta) \\
\neg(\alpha \lor \beta) = \neg\alpha \land \neg\beta \\
\neg(\alpha \land \beta) = \neg\alpha \lor \neg\beta \\
\neg(\phi \rightarrow \alpha/\beta) = \phi \rightarrow \neg\alpha/\neg\beta
\]

Let $s$ be a state. For a formula $\phi \in \Phi$, we write $M, s \models \phi$ or $s \models \phi$ to mean $s \in l(\phi)$. $\pi \models \phi$ iff $s_0 \models \phi$. Also, for an action $\alpha \in \mathcal{A}$, we write $M, s \models_a \alpha$ or $s \models_a \alpha$ to mean that there exists $s' \in W$, $s r(\alpha) s'$. The semantics definition of $\models_a$ on actions can justify the definition of the binary relation $r$ on the complex actions. The semantics of each syntactic formula quoted above over the structure $M = \langle W, r, l, fj, pr \rangle$ is:
\( s \models \top \)
\( s \not\models \bot \)
\( s \models p \quad \text{iff} \quad s \in l(p) \)
\( s \models \neg \phi \quad \text{iff} \quad s \not\models \phi \)
\( s \models \phi \land \psi \quad \text{iff} \quad s \models \phi \quad \text{and} \quad s \models \psi \)
\( s \models \phi \lor \psi \quad \text{iff} \quad s \models \phi \quad \text{or} \quad s \models \psi \)
\( s \models \phi \rightarrow \psi \quad \text{iff} \quad s \models \phi \quad \Rightarrow \quad s \models \psi \)
\( s \models \phi \leftrightarrow \psi \quad \text{iff} \quad s \models \phi \rightarrow \psi \quad \text{and} \quad s \models \psi \rightarrow \phi \)
\( s \models X\phi \quad \text{iff} \quad s' \models \phi \), where \( s' \in W \) and \( s < o s' \)
\( s \models G\phi \quad \text{iff} \quad \forall s' \in W \text{ s.t. } s \leq o s', \ s' \models \phi \)
\( s \models F\phi \quad \text{iff} \quad \exists s' \in W \text{ s.t. } s \leq o s', \ s' \models \phi \)
\( s \models [\alpha]\phi \quad \text{iff} \quad \forall s' \in W, \ s r(\alpha) s' \Rightarrow s' \models \phi \)
\( s \models (\alpha)\phi \quad \text{iff} \quad s \models \neg [\alpha] \neg \phi \)
\( s \models \phi \rightarrow \alpha/\beta \quad \text{iff} \quad (s \models \phi \quad \text{and} \quad s \models_a \alpha) \quad \text{or} \quad (s \models \neg \phi \quad \text{and} \quad s \models_a \beta) \)
\( s \models (B_{epr}(jp))\alpha \quad \text{iff} \quad s \models G(pr(jp) \rightarrow O(\alpha; fj(jp); T?)/T?) \)
\( s \models (A_{fpr}(jp))\alpha \quad \text{iff} \quad s \models G(pr(jp) \rightarrow O((fj(jp)); \alpha; T?)/T?) \)
\( s \models (I_{opr}(jp))\alpha \quad \text{iff} \quad s \models G(pr(jp) \rightarrow O((Ifj(jp)); \alpha; T?)/T?) \)
\( s \models (A_{ipr}(jp))\alpha_1 \alpha_2 \quad \text{iff} \quad s \models (B_{epr}(jp))\alpha_1 \quad \text{and} \quad s \models (A_{fpr}(jp))\alpha_2 \)
\( s \models_a \top \) \iff \( s \models \top \)
\( s \models_a \bot \) \iff \( s \models \bot \)
\( s \models_a b \) \iff \( \exists s' \in W, s \, r(b) \, s' \) where \( b \) is an atomic action
\( s \models_a V \) \iff \( \exists s' \in W, s \, r(V) \, s' \)
\( s \models_a \phi \) \iff \( s \models \phi \)
\( s \models_a \alpha; \beta \) \iff \( \exists s', s'' \in W, s \, r(\alpha) \, s' \) and \( s' \, r(\beta) \, s'' \)
\( s \models_a \alpha \lor \beta \) \iff \( \exists s' \in W, s \, r(\alpha) \, s' \) or \( s \, r(\beta) \, s' \)
\( s \models_a \alpha \land \beta \) \iff \( \exists s' \in W, s \, r(\alpha) \, s' \) and \( s \, r(\beta) \, s' \) \( n \) times
\( s \models_a \alpha^* \) \iff \( \exists n \geq 0, s \models_a \overline{\alpha; \alpha; \ldots; \alpha} \)
\( s \models_a [\alpha] \beta \) \iff \( (s \models_a \neg \alpha) \) or \( (s \models_a \alpha; \beta) \)
\( s \models_a \langle \alpha \rangle \beta \) \iff \( s \models_a \neg [\alpha] \neg \beta \)
\( s \models_a I \alpha \) \iff \( s \models_a [\alpha] V \)
\( s \models_a P \alpha \) \iff \( s \models_a I \neg \alpha \)
\( s \models_a O \alpha \) \iff \( s \models_a I \neg \alpha \)
\( s \models_a O_{s^a} \alpha \) \iff \( (\exists s' \in W, s \, s' \leq_o s' \, s'' \leq_o s'' \, \models_o O \alpha) \)
\( \) and \( \forall \, s'''' \in W, s'''' \leq_o s'''', s''' \models_a I \alpha \)

\textbf{Remark 4.2.} The effect of the action \( V \) depends on the specifier. For example, a punishment action \( V \) could lead to a punishment state (which will be defined by the specifier).

\textbf{Definition 4.5.} Let \( \equiv_a \) a binary relation defined on a set of actions \( \mathcal{A} \) such that \( \alpha \equiv_a \beta \) \iff \( \forall s \in W, s \models_a \alpha \leftrightarrow s \models_a \beta \). \( \equiv_a \) is an equivalence relation.

It is easy to see that \( \equiv_a \) is reflexive, symmetric and transitive; so it is an equivalence relation. We write \( \alpha \equiv_a \beta \) to means that the actions \( \alpha \) and \( \beta \) are equivalents (equality of actions).

\textbf{Theorem 4.1.} \( 1 \) \( [\alpha] (\beta \lor \gamma) \equiv_a [\alpha] \beta \land [\alpha] \gamma \)
\( 2 \) \( [\alpha] (\beta \lor \gamma) \equiv_a [\alpha] \beta \lor [\alpha] \gamma \)
\( 3 \) \( [\alpha \land \beta] \lor \gamma \rightarrow [\alpha] \lor [\beta] \lor \gamma \)
\( 4 \) \( [\alpha \lor \beta] \lor \gamma \equiv_a [\alpha] \lor [\beta] \lor \gamma \)
\( 5 \) \( [\alpha; \beta] \lor \gamma \equiv_a [\alpha] [\beta] \lor \gamma \)
Proof. (3)

\[
s \Vdash_a [\alpha \land \beta] \gamma \iff s \Vdash_a \neg(\alpha \land \beta) \text{ or } s \Vdash_a (\alpha \land \beta); \gamma
\]

\[
s \Vdash_a \neg(\alpha \land \beta) \Rightarrow s \Vdash_a \neg \alpha \text{ or } s \Vdash_a \neg \beta
\]

\[
\Rightarrow s \Vdash_a [\alpha] \gamma \text{ or } s \Vdash_a [\beta] \gamma
\]

(because of the semantics definition of \([\alpha] \beta\) given above)

\[
\Rightarrow s \Vdash_a [\alpha] \gamma \lor [\beta] \gamma
\]

\[
s \Vdash_a (\alpha \land \beta); \gamma \Rightarrow \exists s', s'' \in W \text{ s.t., } s \rho(\alpha \land \beta) s' \text{ and } s' \rho(\gamma) s''
\]

\[
\Rightarrow \exists s', s'' \in W \text{ s.t., } s \rho(\alpha) s' \text{ and } s \rho(\beta) s' \text{ and } s' \rho(\gamma) s''
\]

\[
\Rightarrow \exists s', s'' \in W \text{ s.t., } s \rho(\alpha) s' \text{ and } s' \rho(\gamma) s''
\]

\[
\Rightarrow s \Vdash_a \alpha; \gamma
\]

\[
\Rightarrow s \Vdash_a [\alpha] \gamma
\]

\[
\Rightarrow s \Vdash_a [\alpha] \gamma \text{ or } s \Vdash_a [\beta] \gamma
\]

\[
\Rightarrow s \Vdash_a [\alpha] \gamma \lor [\beta] \gamma
\]

whence, \([\alpha \land \beta] \gamma \rightarrow [\alpha] \gamma \lor [\beta] \gamma \]

The proofs of (1), (2), (4), (5) are done in the same manner of those of (3iii), (3iv), (3vi), (3vii) of the section 4.6.1, respectively. \(\square\)

**Theorem 4.2.** (6) \(I(\alpha; \beta) \equiv_a [\alpha] I(\beta)\)

(7) \(I(\alpha \land \beta) \rightarrow I(\alpha) \lor I(\beta)\)

(8) \(I(\alpha \lor \beta) \equiv_a I(\alpha) \land I(\beta)\)

(9) \(O(\alpha \land \beta) \equiv_a O(\alpha) \land O(\beta)\)

(10) \(O(\alpha \lor \beta) \rightarrow O(\alpha) \lor O(\beta)\)

(11) \(O(\alpha; \beta) \equiv_a O(\alpha) \land [\alpha] O(\beta)\)

(12) \(P(\alpha; \beta) \equiv_a (\alpha) P(\beta)\)

(13) \(P(\alpha) \land P(\beta) \rightarrow P(\alpha \land \beta)\)

(14) \(P(\alpha \lor \beta) \equiv_a P(\alpha) \lor P(\beta)\)
Proof. (6)

\[ s \models_a I(\alpha; \beta) \iff s \models_a [\alpha; \beta]V \]
\[ \iff s \models_a [\alpha][\beta]V \]
\[ \iff s \models_a [\alpha]I\beta \]

whence, \( I(\alpha; \beta) \equiv_a [\alpha]I\beta \quad \square \)

(7)

\[ s \models_a I(\alpha \land \beta) \iff s \models_a [\alpha \land \beta]V \]
\[ \rightarrow s \models_a [\alpha]V \lor [\beta]V \]
\[ \rightarrow s \models_a I\alpha \lor I\beta \]

whence, \( I(\alpha \land \beta) \rightarrow I\alpha \lor I\beta \quad \square \)

(8)

\[ s \models_a I(\alpha \lor \beta) \iff s \models_a [\alpha \lor \beta]V \]
\[ \iff s \models_a [\alpha]V \land [\beta]V \]
\[ \iff s \models_a I\alpha \land I\beta \]

whence, \( I(\alpha \lor \beta) \equiv_a I\alpha \land I\beta \quad \square \)

(9)

\[ s \models_a O(\alpha \land \beta) \iff s \models_a I-(\alpha \land \beta) \]
\[ \iff s \models_a I(-\alpha \lor -\beta) \]
\[ \iff s \models_a I-\alpha \land I-\beta \]
\[ \iff s \models_a O\alpha \land O\beta \]

whence, \( O(\alpha \land \beta) \equiv_a O\alpha \land O\beta \quad \square \)

(10)

\[ s \models_a O(\alpha \lor \beta) \iff s \models_a I-(\alpha \lor \beta) \]
\[ \iff s \models_a I(-\alpha \land -\beta) \]
\[ \rightarrow s \models_a I-\alpha \lor I-\beta \]
\[ \rightarrow s \models_a O\alpha \lor O\beta \]

whence, \( O(\alpha \lor \beta) \rightarrow O\alpha \lor O\beta \quad \square \)
(11) 
\[
\begin{align*}
s \models_a O(\alpha; \beta) & \iff s \models_a \neg(\alpha; \beta) \\
& \iff s \models_a I(\neg\alpha \lor (\alpha; \neg\beta)) \\
& \iff s \models_a [\neg\alpha \lor (\alpha; \neg\beta)]V \\
& \iff s \models_a [\neg\alpha]V \land [\alpha; \neg\beta]V \\
& \iff s \models_a [\neg\alpha]V \land [\alpha][\neg\beta]V \\
& \iff s \models_a I\neg\alpha \land [\alpha]I\neg\beta \\
& \iff s \models_a O\alpha \land [\alpha]O\beta \\
\text{whence, } O(\alpha; \beta) & \equiv_a O\alpha \land [\alpha]O\beta \quad \square
\end{align*}
\]

(12) 
\[
\begin{align*}
s \models_a P(\alpha; \beta) & \iff s \models_a \neg(I(\alpha; \beta)) \\
& \iff s \models_a \neg[\alpha; \beta]V \\
& \iff s \models_a \neg([\alpha][\beta]V) \\
& \iff s \models_a (\alpha)(\neg\beta) \\
& \iff s \models_a (\alpha)P\beta \\
\text{whence, } P(\alpha; \beta) & \equiv_a (\alpha)P\beta \quad \square
\end{align*}
\]

(13) 
We have \(I(\alpha \land \beta) \rightarrow I\alpha \lor I\beta\) (by theorem 4.2(7))

thus \(\neg(I\alpha \lor I\beta) \rightarrow \neg I(\alpha \land \beta)\)

thus \((\neg I\alpha) \land (\neg I\beta) \rightarrow \neg I(\alpha \land \beta)\)

whence, \(P\alpha \land P\beta \rightarrow P(\alpha \land \beta) \quad \square\)

(14) 
\[
\begin{align*}
s \models_a P(\alpha \lor \beta) & \iff s \models_a \neg(I(\alpha \lor \beta)) \\
& \iff s \models_a \neg(I\alpha \land I\beta) \\
& \iff s \models_a \neg(I\alpha) \lor (\neg I\beta) \\
& \iff s \models_a P\alpha \lor P\beta \\
\text{whence, } P(\alpha \lor \beta) & \equiv_a P\alpha \lor P\beta \quad \square
\end{align*}
\]
4.4 Satisfiability and Validity

Φ is the set of all propositions of $L_A$.

**Definition 4.6.** Let $M = \langle W, r, l, fj, pr \rangle$ be a model, $s \in W$, and $\phi \in \Phi$. We write $M, s \models \phi$ if for every execution path $\pi = s_0 s_1 \ldots$ of $M$ starting at $s$, we have $\pi \models \phi$.

**Definition 4.7.** Let $\phi \in \Phi$ be a proposition. If there exists a model $M = \langle W, r, l, fj, pr \rangle$ and a state $s \in W$ such that $M, s \models \phi$, we say that $\phi$ is satisfiable.

**Definition 4.8.** Let $M = \langle W, r, l, fj, pr \rangle$ be a model, and $\phi \in \Phi$. $\phi$ is valid w.r.t $M$ denoted by $M \models \phi$ if $M, s \models \phi$ for all $s \in W$. $M, s \models \phi$ if $s \models \phi$ in the model $M$.

**Definition 4.9.** Let $\phi \in \Phi$. $\phi$ is valid (denoted by $\models \phi$) if $M \models \phi$ for all models $M$.

**Definition 4.10.** We say that two $L_A$ formulas $\phi$ and $\psi$ are semantically equivalent, or simply equivalent, writing $\phi \equiv \psi$ if for all models $M$ and all paths $\pi$ in $M$, $\pi \models \phi$ iff $\pi \models \psi$. $\phi \equiv \psi$ means that the formulae $\phi$ and $\psi$ are equivalent by definition (i.e., the member of the left side is defined by the member of the right side).

**Remark 4.3.** Satisfiability and validity are dual in the same sense that $\exists$ and $\forall$ are dual and $[]$ and $\langle \rangle$ are dual: a proposition is valid if only if its negation is not satisfiable.

4.5 Axiom System and Inference Rules

In this section, we present axioms and inference rules of the logic $L_A$. Let $np$ be a pointcut containing joinpoints $jp_i$, $i = 1, 2$; let $p_i = pr(jp_i)$ for $i = 1, 2$.

1. If $np = jp_1 || jp_2$ then,

   (i) $(B_c pr(np)) \alpha \overset{\text{def}}{=} (B_c p_1) \alpha \land (B_c p_2) \alpha$

   (ii) $(A_f pr(np)) \alpha \overset{\text{def}}{=} (A_f p_1) \alpha \land (A_f p_2) \alpha$

   (iii) $(A_r pr(np)) \alpha \overset{\text{def}}{=} (A_r p_1) \alpha \land (A_r p_2) \alpha$

   (iv) $(I_o pr(np)) \alpha \overset{\text{def}}{=} (I_o p_1) \alpha \land (I_o p_2) \alpha$

2. If $np = jp_1 && jp_2$ then,
(i) \((B_e \text{pr}(np))\alpha \overset{\text{def}}{=} \begin{cases} (B_e p_2)\alpha & \text{if } p_1 \leq p_2; \\ (B_e p_1)\alpha & \text{if } p_2 \leq p_1. \end{cases}\)

(ii) \((A_f \text{pr}(np))\alpha \overset{\text{def}}{=} \begin{cases} (A_f p_2)\alpha & \text{if } p_1 \leq p_2; \\ (A_f p_1)\alpha & \text{if } p_2 \leq p_1. \end{cases}\)

(iii) \((A_r \text{pr}(np))\alpha \overset{\text{def}}{=} \begin{cases} (A_r p_2)\alpha & \text{if } p_1 \leq p_2; \\ (A_r p_1)\alpha & \text{if } p_2 \leq p_1. \end{cases}\)

(iv) \((I_o \text{pr}(np))\alpha \overset{\text{def}}{=} \begin{cases} (I_o p_2)\alpha & \text{if } p_1 \leq p_2; \\ (I_o p_1)\alpha & \text{if } p_2 \leq p_1. \end{cases}\)

3. (i) Axioms for linear temporal logic

(ii) \([\alpha](\phi \rightarrow \psi) \rightarrow [\alpha]\phi \rightarrow [\alpha]\psi\)

(iii) \([\alpha](\phi \land \psi) \equiv [\alpha]\phi \land [\alpha]\psi\)

(iv) \([\alpha]\phi \lor [\alpha]\psi \rightarrow [\alpha](\phi \lor \psi)\)

(v) \([\alpha]\phi \lor [\beta]\phi \rightarrow [\alpha \land \beta]\phi\)

(vi) \([\alpha \lor \beta]\phi \equiv [\alpha]\phi \land [\beta]\phi\)

(vii) \([\alpha; \beta]\phi \equiv [\alpha][\beta]\phi\)

(viii) \([\psi?]\phi \equiv (\psi \rightarrow \phi)\)

(ix) \(\phi \land [\alpha][\alpha^*]\phi \equiv [\alpha^*]\phi\)

(x) \(\phi \land [\alpha^*](\phi \rightarrow [\alpha]\phi) \rightarrow [\alpha^*]\phi\)

4. Inference rules

(i) \(\phi, \phi \rightarrow \psi \over \psi\)

(ii) \(\phi \over [\alpha]\phi\)

(iii) \(\phi \over [\alpha]\phi \rightarrow [\alpha]\psi\)

(iv) \(\phi \over (\alpha)\phi \rightarrow (\alpha)\psi\)
4.6 Soundness and Completeness

We write $\vdash \varphi$ to indicate that the formula $\varphi$ is provable in the proof system $\vdash$. $\not\vdash \varphi$ means that $\varphi$ is not provable in the proof system $\vdash$.

**Remark 4.4.** It is important to keep in mind that is the *ought to* do approach we use in our logic. A proposition or formula of $L_A$ is not an action and an action is not a proposition. Deontic operators are uniquely defined over actions. The domain of these operators is the set $\mathcal{A}$ of actions and their range is included in $\mathcal{A}$. It has been shown in the literature [58] that the *ought-to-do* approach allows to avoid or to resolve some classical paradoxes of the standard deontic logic.

**Remark 4.5.** It is also important to notice that we use the dynamic logic style to define deontic operators. For example, $I\alpha = [\alpha]V$, where $I$ is the forbidden operator of the deontic logic, $\alpha$ is an action; in this equality, $V$ is not a proposition (such as in dynamic logic), but an action specifying what to do in the case one perform the action $\alpha$. The advantage of this is the fact that $V$ could be a reparation or a punishment action in the case of a violation. This will allow us to build our prevention policy for undesirable aspect interactions. This kind of specification could also be use to specify fault tolerant systems.

Because of the above remarques, some proofs are similar to ones of dynamic logic in [43] or ones of modal logic in [12]. We will indicate each time it is the case.

**Definition 4.11. Soundness and Completeness**

A deductive system $\vdash$ is said to be sound with respect to a semantics $\models$ if for all sentences $\varphi$, $\vdash \varphi \Rightarrow \models \varphi$.
A deductive system $\vdash$ is said to be complete with respect to a semantics $\models$ if for all $\varphi$, $\models \varphi \Rightarrow \vdash \varphi$, i.e., all valid formulae are theorems.

### 4.6.1 Soundness

Soundness means that every theorem is valid, i.e., every time we prove something, we have guaranty that it is true in every model. To establish that an axiom system $L$ is sound, one establish as that every line of a proof in $L$ is valid in all models, and hence all theorems are valid. To do this, it is enough to show that all axioms are valid, and the rules preserve validity (i.e., if premises of a rule are valid, then the conclusion of that rule is also valid).
Definition 4.12. Soundness of an Inference Rule
An inference rule \( \frac{\phi_1, \phi_2, \ldots, \phi_n}{\psi} \) is sound iff the validity of the premises \( \phi_1, \phi_2, \ldots, \phi_n \) implies that the conclusion \( \psi \) is valid.

Theorem 4.3. The above list of axioms and rules (defined in 4.5) constitutes a sound Hilbert-style deductive system for \( L_A \).

Definition 4.13. Let \( \alpha \) be an action. We define the action \( \alpha^n \) for \( n \in \mathbb{N} \) as follows:

\[
\alpha^0 = \top
\]
\[
\alpha^{n+1} = \alpha^n; \alpha
\]

We have \( s \models_a \alpha^* \iff \exists n \geq 0, s \models_a \alpha^n \) (that is, \( r(\alpha^*) \overset{\text{def}}{=} \bigcup_{n \geq 0} r(\alpha^n) \)).

Proof. (of theorem 4.3) Let \( M = \{W, r, l, f, p\} \) be a model of \( L_A \), \( s \in W \), \( \phi \) belongs to the set of the above axioms. We want to show \( M, s \models \phi \). The axioms-definitions 1 (i to iv) and 2 (i to iv) are definitions, i.e., the left side of each of these axioms is defined by the right side. The proof of (3i) can be found in any linear temporal logic book.

(3ii) (\( \rightarrow \))

We know that \( [\alpha](\varphi \rightarrow \psi) \equiv [\alpha](\neg \varphi \lor \psi) \)

\[
[\alpha] \neg \varphi \lor [\alpha] \psi \rightarrow [\alpha](\neg \varphi \lor \psi)
\]

because of 3iv

and \( [\alpha] \neg \varphi \rightarrow \neg ( [\alpha] \varphi ) \)

thus \( [\alpha] \neg \varphi \lor [\alpha] \psi \rightarrow \neg ( [\alpha] \varphi ) \lor [\alpha] \psi \)

\equiv [\alpha] \varphi \rightarrow [\alpha] \psi

whence, \( [\alpha](\varphi \rightarrow \psi) \rightarrow ([\alpha] \varphi \rightarrow [\alpha] \psi) \)

(3iii)

\[
s \models [\alpha](\varphi \land \psi) \iff \forall s' \in W, s \models r(\alpha) s' \Rightarrow (s' \models \varphi \land \psi)
\]

\iff \forall s' \in W, s \models r(\alpha) s' \Rightarrow (s' \models \varphi \land \psi)

\iff (\forall s' \in W, s \models r(\alpha) s' \Rightarrow s' \models \varphi) \land (\forall s' \in W, s \models r(\alpha) s' \Rightarrow s' \models \psi)

\iff s \models [\alpha] \varphi \land s \models [\alpha] \psi

whence, \( [\alpha](\varphi \land \psi) \equiv [\alpha] \varphi \land [\alpha] \psi \)
(3iv)

\[ s \models [\alpha] \phi \lor [\alpha] \psi \iff s \models [\alpha] \phi \text{ or } s \models [\alpha] \psi \]

\[ \iff (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow s' \models \phi) \text{ or } (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow s' \models \psi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow s' \models \phi \lor s' \models \psi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow s \models [\alpha](\phi \lor \psi) \]

whence, \[ [\alpha] \phi \lor [\alpha] \psi \rightarrow [\alpha](\phi \lor \psi) \]

(3v)

\[ s \models [\alpha] \phi \lor [\beta] \phi \iff s \models [\alpha] \phi \text{ or } s \models [\beta] \phi \]

\[ \iff (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow s' \models \phi) \text{ or } (\forall s' \in W, s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow s' \models \phi) \text{ or } (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

\[ \Rightarrow (\forall s' \in W, s \mathcal{r}(\alpha) s' \Rightarrow (s \mathcal{r}(\beta) s' \Rightarrow s' \models \phi) \]

whence, \[ [\alpha] \phi \lor [\beta] \phi \rightarrow [\alpha \land \beta] \phi \]
(3vi)

\[ s \models [\alpha \lor \beta] \varphi \iff \forall s' \in W, s \ r(\alpha \lor \beta) \ s' \implies s' \models \varphi \]
\[ \iff \forall s' \in W, s \ r(\alpha) \ s' \lor s \ r(\beta) \ s' \implies s' \models \varphi \]
\[ \iff \forall s' \in W, s \ r(\alpha) \ s' \lor s \ r(\beta) \ s' \implies s' \models \varphi \]
\[ \iff \forall s' \in W, s \ r(\alpha) \ s' \land s \ r(\beta) \ s' \implies s' \models \varphi \]
\[ \iff \forall s' \in W, (s \ r(\alpha) \ s' \land s' \models \varphi) \land (s \ r(\beta) \ s' \implies s' \models \varphi) \]
\[ \iff \forall s' \in W, (s \ r(\alpha) \ s' \implies s' \models \varphi) \land (s \ r(\beta) \ s' \implies s' \models \varphi) \]
\[ \iff s \models [\alpha] \varphi \land s \models [\beta] \varphi \]
\[ \iff s \models [\alpha] \varphi \land [\beta] \varphi \]
whence, \ [\alpha \lor \beta] \varphi \equiv [\alpha] \varphi \land [\beta] \varphi \square

(3vii)

\[ s \models [\alpha; \beta] \varphi \iff \forall s'' \in W, s \ r(\alpha; \beta) \ s'' \implies s'' \models \varphi \]
\[ \iff \forall s'' \in W, (\exists \ s' \in W, s \ r(\alpha) \ s' \text{ and } s' \ r(\beta) \ s'') \implies s'' \models \varphi \]
\[ \iff \forall s'' \in W, (\exists \ s' \in W, s \ r(\alpha) \ s' \text{ and } s' \ r(\beta) \ s'') \lor s'' \models \varphi \]
\[ \iff \forall s'' \in W, (\forall s' \in W, s \ r(\alpha) \ s' \lor \ s' \ r(\beta) \ s'') \lor s'' \models \varphi \]
\[ \iff \forall s'' \in W, \forall s' \in W, s \ r(\alpha) \ s' \lor \ (s' \ r(\beta) \ s'') \lor s'' \models \varphi \]
\[ \iff \forall s' \in W, s \ r(\alpha) \ s' \lor (\forall s'' \in W, s' \ r(\beta) \ s'') \implies s'' \models \varphi \]
\[ \iff \forall s' \in W, s \ r(\alpha) \ s' \lor s' \models [\beta] \varphi \]
\[ \iff \forall s' \in W, s \ r(\alpha) \ s' \lor s' \models [\beta] \varphi \]
\[ \iff s \models [\alpha] [\beta] \varphi \]
whence, \ [\alpha; \beta] \varphi \equiv [\alpha] [\beta] \varphi \square

(3viii)

\[ s \models [\psi]? \varphi \iff \forall s' \in W, s \ r(\psi)? \ s' \implies s' \models \varphi \]
\[ \iff \forall s' \in W, (s = s' \text{ and } s \models \psi) \implies s' \models \varphi \]
\[ \iff s \models \psi \implies s \models \varphi \]
\[ \iff s \models \psi \implies \varphi \]
whence, \ [\psi]? \varphi \equiv \psi \implies \varphi \quad \square
(3ix)($\rightarrow$)

\[ s \models \varphi \land [\alpha][\alpha^*]\varphi \Rightarrow s \models [\alpha][\alpha^*]\varphi \quad \text{and} \quad s \models \varphi \]

\[ \Rightarrow s \models [\alpha^*]\varphi \text{ hence, } \varphi \land [\alpha][\alpha^*]\varphi \rightarrow [\alpha^*]\varphi \]

(3ix)($\leftarrow$)

Suppose \( s \models [\alpha^*]\varphi \)

Let \( t \in W \) s.t. \( s \models r(\alpha) t \)

\[ s \models [\alpha^*]\varphi \Rightarrow (\forall s' \in W, s \models r(\alpha^*) s' \Rightarrow s' \models \varphi) \]

\[ t \models r(\alpha^*) s' \Rightarrow s \models r(\alpha^*) s' \Rightarrow s' \models \varphi \]

so, \( \forall t \in W, s \models r(\alpha) t \Rightarrow t \models [\alpha^*]\varphi \)

\[ \Rightarrow s \models [\alpha][\alpha^*]\varphi. \text{ By definition, we have } s \models r(\alpha) s; \text{ hence, } s \models \varphi \]

\[ \Rightarrow s \models \varphi \land [\alpha][\alpha^*]\varphi \text{ hence, } [\alpha^*]\varphi \rightarrow \varphi \land [\alpha][\alpha^*]\varphi \]

(3x)

Suppose \( s \models \varphi \) and \( s \models [\alpha^*](\varphi \rightarrow [\alpha]\varphi) \)

We want to show that \( s \models [\alpha^*]\varphi \)

We show by induction that \( s \models [\alpha^n]\varphi, \forall n \geq 0 \)

- \( n = 0; s \models \varphi \Rightarrow s \models [\alpha^0]\varphi \)
- Assume that \( s \models [\alpha^n]\varphi. \) We want to show that \( s \models [\alpha^{n+1}]\varphi \)

Let \( s' \in W \) s.t. \( s \models r(\alpha^{n+1}) s' \)

\[ \Rightarrow \exists t \in W, s \models r(\alpha^n) t \text{ and } t \models r(\alpha) s' \]

\[ s \models [\alpha^n]\varphi \Rightarrow (t \models \varphi \Rightarrow t \models [\alpha]\varphi) \]

(by hypothesis)

\[ s \models [\alpha^n]\varphi \Rightarrow (t \models \varphi \Rightarrow t \models [\alpha]\varphi) \]

So, \( t \models r(\alpha) s' \Rightarrow s' \models \varphi \) thus, \( s \models r(\alpha^{n+1}) s' \Rightarrow s' \models \varphi \)

\[ \Rightarrow s \models [\alpha^{n+1}]\varphi \text{ whence, } s \models [\alpha^n]\varphi, \forall n \geq 0 \]

whence, \( s \models [\alpha^*]\varphi \) and thus, \( \varphi \land [\alpha^*](\varphi \rightarrow [\alpha]\varphi) \rightarrow [\alpha^*]\varphi \)

(4i)

\[ (s \models \varphi \text{ and } s \models \varphi \rightarrow \psi) \Rightarrow s \models \varphi \text{ and } (s \models \varphi \Rightarrow s \models \psi) \]

\[ \Rightarrow s \models \psi \text{ whence, } (\models \varphi \text{ and } \models \varphi \rightarrow \psi) \Rightarrow \models \psi \]
(4ii)

\[ \models \varphi \Rightarrow \models [\alpha] \varphi \] is obvious because \( \models \varphi \) asserts that \( \varphi \) holds at all times, whence no matter \( \alpha \) might take us, \( \varphi \) will be true there.

It is worth to note that the implication \( \varphi \rightarrow [\alpha] \varphi \) is not valid, however, because the truth of \( \varphi \) at the present moment is no guarantee of its truth after performing \( \alpha \).

(4iii)

\[ \models \varphi \rightarrow \psi \Rightarrow \models [\alpha][(\varphi \rightarrow \psi)] \quad (4ii) \]

\[ \Rightarrow \models [\alpha] \varphi \rightarrow [\alpha] \psi \quad \text{whence,} \quad \models \varphi \rightarrow \psi \Rightarrow \models [\alpha] \varphi \rightarrow [\alpha] \psi \] □

(4iv)

\[ s \models \langle \alpha \rangle \varphi \Rightarrow s \models [\alpha] \neg \varphi \]

\[ \Rightarrow \neg (\forall s' \in W, s r(\alpha) s' \Rightarrow s' \models \neg \varphi) \]

\[ \Rightarrow \neg (\forall s' \in W, s r(\alpha) s' \text{ or } s' \models \neg \varphi) \]

\[ \Rightarrow \exists s' \in W, s r(\alpha) s' \text{ and } s' \models \varphi \]

\[ \Rightarrow \exists s' \in W, s r(\alpha) s' \Rightarrow s' \models \psi \quad \text{(by hypothetis)} \]

\[ \Rightarrow (\forall s' \in W, s r(\alpha) s' \text{ or } s' \models \neg \psi) \]

\[ \Rightarrow (\forall s' \in W, s r(\alpha) s' \Rightarrow s' \models \neg \psi) \]

\[ \Rightarrow s \models [\alpha] \neg \psi \]

\[ \Rightarrow s \models \langle \alpha \rangle \psi \quad \text{whence,} \quad \models \varphi \rightarrow \psi \Rightarrow \models \langle \alpha \rangle \varphi \rightarrow \langle \alpha \rangle \psi \] □

4.6.2 Completeness

Completeness means that our proof system is strong enough to prove everything that is valid. It is typically proved by contraposition, i.e.,

\[ \not\models \varphi \Rightarrow \pi \not\models \varphi \]

\[ \Rightarrow \neg \varphi \text{ has a model} \]
The completeness problem is thus a model building task. We will use similar techniques from [43] to prove the completeness of our logic.

**Definition 4.14.** _Consistency and Refutability_

A formula \( \varphi \) is _refutable_ if \( \lnot \varphi \) is a theorem; that is, if \( \vdash \lnot \varphi \). A set \( \Phi \) of sentences is refutable if some finite conjunction of elements of \( \Phi \) is refutable. \( \varphi \) or \( \Phi \) is _consistent_ if it is not refutable.

**Lemma 4.1.** Let \( \Sigma \) be a set of formulas of \( L_A \). Then

1. \( \Sigma \) is consistent iff either \( \Sigma \cup \{ \varphi \} \) is consistent or \( \Sigma \cup \{ \lnot \varphi \} \) is consistent;

2. if \( \Sigma \) is consistent, then \( \Sigma \) is contained in a maximal consistent set;

3. if \( \Sigma \) is a maximal consistent set of formulas, then:
   
   (i) \( \Sigma \) contains all theorems of \( L_A \);
   
   (ii) if \( \varphi \in \Sigma \) and \( \varphi \rightarrow \psi \in \Sigma \), then \( \psi \in \Sigma \);

   (iii) \( \varphi \lor \psi \in \Sigma \) iff \( \varphi \in \Sigma \) or \( \psi \in \Sigma \);

   (iv) \( \varphi \land \psi \in \Sigma \) iff \( \varphi \in \Sigma \) and \( \psi \in \Sigma \);

   (v) \( \varphi \in \Sigma \) iff \( \lnot \varphi \not\in \Sigma \);

   (vi) \( \bot \not\in \Sigma \).

**Remark 4.6.** \( \Sigma \) is contained in a maximal consistent set \( \hat{\Sigma} \) means a consistent set such that \( \Sigma \subseteq \hat{\Sigma} \) and any proper superset of \( \hat{\Sigma} \) is refutable.

**Proof.** (1)

\( \lnot \) is trivial

\( \Rightarrow \) Suppose \( \Sigma \) is consistent and \( (\Sigma \cup \{ \varphi \} \) and \( \Sigma \cup \{ \lnot \varphi \} \) are refutable

\( \Sigma \cup \{ \varphi \} \) and \( \Sigma \cup \{ \lnot \varphi \} \) refutable \( \Rightarrow \exists \{ \varphi_1, \varphi_2, \ldots, \varphi_n \} \subseteq \Sigma \) and \( \{ \psi_1, \psi_2, \ldots, \psi_n \} \subseteq \Sigma \) s.t.

\( \varphi_1 \land \varphi_2 \land \ldots \land \varphi_n \land \varphi \) and \( \psi_1 \land \psi_2 \land \ldots \land \psi_n \land \lnot \varphi \) are refutable

\( \Sigma \) consistent \( \Rightarrow \ varphi_1 \land \varphi_2 \land \ldots \land \varphi_n \) and \( \psi_1 \land \psi_2 \land \ldots \land \psi_n \) consistent

\( \varphi_1 \land \varphi_2 \ldots \land \varphi_n \) consistent and \( \varphi_1 \land \varphi_2 \ldots \land \varphi_n \land \varphi \) refutable \( \Rightarrow \varphi \) refutable

\( \psi_1 \land \psi_2 \ldots \land \psi_n \) consistent and \( \psi_1 \land \psi_2 \ldots \land \psi_n \land \lnot \varphi \) refutable \( \Rightarrow \lnot \varphi \) refutable

\( \Rightarrow \varphi \) refutable and \( \lnot \varphi \) refutable; what is absurd

whence \( \Sigma \) consistent \( \Rightarrow \Sigma \cup \{ \varphi \} \) or \( \Sigma \cup \{ \lnot \varphi \} \) is consistent \( \Box \)
We want to show $\Sigma$ consistent $\Rightarrow$ there is a maximal consistent set $\hat{\Sigma}$ such that $\Sigma \subseteq \hat{\Sigma}$. The construction of $\hat{\Sigma}$ is the same as in [43] p. 80. or in [12], p. 197 because our logic is a modal logic. □

(3i)

We want to show $\Sigma$ contains all theorems of $\mathcal{L}_a$. Let $\varphi$ be a theorem of $\mathcal{L}_a$. Suppose $\varphi \notin \Sigma$

$\Rightarrow \Sigma \cup \{\varphi\}$ is refutable (because $\Sigma$ is maximal)

$\Rightarrow \exists \varphi_1, \varphi_2, ..., \varphi_n \in \Sigma$ s.t. $\varphi_1 \land \varphi_2 ... \land \varphi_n \land \varphi$ is refutable

$\Sigma$ consistent $\Rightarrow \varphi_1 \land \varphi_2 ... \land \varphi_n$ consistent

$\varphi_1 \land \varphi_2 ... \land \varphi_n$ consistent and $\varphi_1 \land \varphi_2 ... \land \varphi_n \land \varphi$ refutable $\Rightarrow \varphi$ refutable

Contradiction since $\varphi$ is a theorem. Whence $\varphi \in \Sigma$. □

(3ii) $\varphi \in \Sigma$ implies $\varphi$ is consistent; this implies $\vdash \varphi$. $\varphi \rightarrow \psi \in \Sigma$ implies $\varphi \rightarrow \psi$ is consistent; this implies $\vdash \varphi \rightarrow \psi$, and so $(\vdash \varphi \Rightarrow \vdash \psi)$. Thus $\vdash \psi$ since we have $\vdash \varphi$. That implies $\psi \in \Sigma$ because of (3i). Whence $\varphi \in \Sigma$ and $\varphi \rightarrow \psi \in \Sigma$ implies $\psi \in \Sigma \square$.

(3iii) $\varphi \lor \psi \in \Sigma$ $\iff \varphi \lor \psi$ is consistent $\iff \vdash \varphi \lor \psi \iff \vdash \varphi$ or $\vdash \psi$ $\iff \varphi \in \Sigma$ or $\psi \in \Sigma$ (because of (3i)). Whence $\varphi \lor \psi \in \Sigma$ $\iff \varphi \in \Sigma$ or $\psi \in \Sigma \square$.

(3iv) $\varphi \land \psi \in \Sigma$ $\iff \varphi \land \psi$ is consistent $\iff \vdash \varphi \land \psi \iff \vdash \varphi$ and $\vdash \psi$ $\iff \varphi \in \Sigma$ and $\psi \in \Sigma$ (3i). Whence $\varphi \land \psi \in \Sigma \iff \varphi \in \Sigma$ and $\psi \in \Sigma \square$.

(3v) $\varphi \in \Sigma$ $\iff \varphi$ is consistent $\iff \vdash \varphi \iff \neg(\vdash \neg \varphi) \iff \neg \varphi \notin \Sigma$. Whence $\varphi \in \Sigma \iff \neg \varphi \notin \Sigma \square$.

(3vi) $\bot \in \Sigma$ implies $\bot$ is consistent; this implies $\vdash \bot$: contradiction. Whence $\bot \notin \Sigma \square$.

**Lemma 4.2.** Let $\Sigma$ and $\Gamma$ be maximal consistent sets of formulas and let $\alpha$ be an action.

The following two statements are equivalent:

(a) For all formulas $\psi$, if $\psi \in \Gamma$ then $(\langle \alpha \rangle \psi) \in \Sigma$. 
(b) For all formulas $\varphi$, if $[\alpha] \varphi \in \Sigma$ then $\varphi \in \Gamma$.

Proof. The proof of this lemma can be found in [43].

**Definition 4.15. Nonstandard Model**

A nonstandard model is any structure $M = \langle W, r, l, f, p, r \rangle$ defined in section 4.3 for which $r(\alpha^*)$ need not be the reflexive transitive closure of $r(\alpha)$, but only a reflexive, transitive binary relation containing $r(\alpha)$ satisfying axioms for $*$; that is,

$$ r(\alpha^*) \overset{\text{def}}{=} \bigcup_{n \geq 0} r(\alpha^n) \quad (4.1) $$

is replaced by the weaker requirement that $r(\alpha^*)$ be reflexive, transitive binary relation containing $r(\alpha)$ such that:

$$ l([\alpha^*] \varphi) = l(\varphi \land [\alpha; \alpha^*] \varphi) $$

$$ l([\alpha^*] \varphi) = l(\varphi \land [\alpha^*](\varphi \to [\alpha] \varphi) $$

Now we construct a nonstandard model $M_n = \langle W_n, r_n, l_n, f, p, r \rangle$.

$$ W_n \overset{\text{def}}{=} \{ \text{maximal consistent sets of formulae of } \mathcal{L}_a \} $$

$$ l_n(\varphi) \overset{\text{def}}{=} \{ s \in W_n / \varphi \in s \} $$

$$ r_n(\alpha) \overset{\text{def}}{=} \{ (s, t) \in W_n \times W_n / \text{for all } \varphi, \text{ if } \varphi \in t, \text{ then } [\alpha] \varphi \in s \} $$

$$ = \{ (s, t) \in W_n \times W_n / \text{for all } \varphi, \text{ if } [\alpha] \varphi \in s, \text{ then } \varphi \in t \} $$

We want to show that $M_n$ is a nonstandard model.

**Lemma 4.3.** (i) $l_n(\varphi \to \psi) = (W_n - l_n(\varphi)) \cup l_n(\psi)$

(ii) $l_n(\bot) = \emptyset$

(iii) $l_n([\alpha] \varphi) = \{ s / \forall t \in W_n, (s, t) \in r_n(\alpha) \Rightarrow t \in l_n(\varphi) \}$

(iv) $l_n(\varphi \to \alpha / \beta) = \{ s \in l_n(\varphi) / \exists t \in W_n, (s, t) \in r_n(\alpha) \} \cup \{ s \in l_n(\neg \varphi) / \exists t \in W_n, (s, t) \in r_n(\beta) \}$
Proof. (i)

\[ s \in l_n(\varphi \rightarrow \psi) \iff \varphi \rightarrow \psi \in s \]

\[ \iff \varphi \in s \Rightarrow \psi \in s \quad \text{by lemma 4.1(3ii)} \]

\[ \iff \varphi \not\in s \text{ or } \psi \in s \]

\[ \iff s \not\in l_n(\varphi) \text{ or } s \in l_n(\psi) \]

\[ \iff s \in W_n - l_n(\varphi) \text{ or } s \in l_n(\psi) \]

\[ \iff s \in (W_n - l_n(\varphi)) \cup l_n(\psi) \]

Whence \[ l_n(\varphi \rightarrow \psi) = (W_n - l_n(\varphi)) \cup l(\psi) \]

(ii) \[ s \in l_n(\bot) \Rightarrow \bot \in s, \text{ absurd. This implies } s \in \emptyset; \text{ and so } l_n(\bot) \subseteq \emptyset. \text{ Whence } l_n(\bot) = \emptyset. \]

(iii) The proof can be found in [43]. \[ \square \]

(iv)

\[ s \in l_n(\varphi \rightarrow \alpha/\beta) \iff \varphi \rightarrow \alpha/\beta \in s \]

\[ \iff (\varphi \rightarrow \alpha) \lor (\neg \varphi \rightarrow \beta) \in s \quad \text{semantics of } \varphi \rightarrow \alpha/\beta \]

\[ \iff \varphi \rightarrow \alpha \in s \text{ or } \neg \varphi \rightarrow \beta \in s \quad \text{by lemma 4.1(3iii)} \]

\[ \iff (s \in l_n(\varphi) / \exists t \in W_n, (s, t) \in r_n(\alpha)) \]

or \[ (s \in l_n(\neg \varphi) / \exists t \in W_n, (s, t) \in r_n(\beta)) \]

\[ \iff s \in \{s \in l_n(\varphi) / \exists t \in W_n, (s, t) \in r_n(\alpha)\} \cup \]

\[ \{s \in l_n(\neg \varphi) / \exists t \in W_n, (s, t) \in r_n(\beta)\} \]

\[ \square \]

Lemma 4.4. (i) \[ r_n(\alpha \lor \beta) = r_n(\alpha) \cup r_n(\beta) \]

(ii) \[ r_n(\alpha; \beta) = \{(s, t) / \exists z \in W_n, (s, z) \in r_n(\alpha) \text{ and } (z, t) \in r_n(\beta)\} \]

(iii) \[ r_n(\varphi?) = \{(s, s) / s \in l_n(\varphi)\} \]
Proof. (i)

\[(s, t) \in r_n(\alpha \lor \beta) \iff \forall \varphi, (\alpha \lor \beta) \varphi \in s \implies \varphi \in t \]

\[(\iff \forall \varphi, (\alpha) \varphi \land (\beta) \varphi \in s \implies \varphi \in t \]

\[(\iff \forall \varphi, (\alpha) \varphi \not\in s \lor (\beta) \varphi \not\in s) \text{ or } \varphi \in t \]

\[(\iff \forall \varphi, (\alpha) \varphi \not\in s \lor \varphi \in t) \text{ or } (\beta) \varphi \not\in s \lor \varphi \in t \]

\[(\iff \forall \varphi, (\alpha) \varphi \in s \implies \varphi \in t) \text{ or } (\beta) \varphi \in s \implies \varphi \in t \]

\[(\iff (s, t) \in r_n(\alpha) \text{ or } (s, t) \in r_n(\beta)) \]

Whence \(r_n(\alpha \lor \beta) = r_n(\alpha) \cup r_n(\beta) \)

(ii) and (iii)

The proofs can be found in [43]. \(\square\)

\(\square\)

**Theorem 4.4.** The model \(M_n\) is a nonstandard model.

Proof. Using lemmas 4.3 and 4.4, the operators \(\rightarrow, \bot, [], ;, \lor, \land\) behave in \(M_n\) as in standard models. Now we want to show that the following properties

\[\varphi \land [\alpha][\alpha^\ast] \varphi \equiv [\alpha^\ast] \varphi\]

\[\varphi \land [\alpha^\ast] (\varphi \rightarrow [\alpha] \varphi) \equiv [\alpha^\ast] \varphi\]

are theorems. But, this is immediate by using inference rule (ii) of the axioms system of \(L_A\). Now, by lemma 4.1 (3i), these theorems are in every maximal consistent set. Thus, this shows that \(M_n\) satisfies conditions in the definition of nonstandard model. \(\square\)

The definition of the nonstandard model \(M_n\) is independent of any particular \(\varphi\). It is a universal model in the sense that every consistent formula is satisfied at some state of \(M_n\).

Our completeness proof uses a generalization of the Fisher-Ladner closure [35] of a proposition. The Fisher-Ladner closure \(FL(\phi)\) of a proposition \(\phi\) of \(L\) is the smallest set of propositions containing \(\phi\) and satisfying the definition of the Fisher-Ladner closure:
\((f_1)\) if \(-\varphi \in FL(\phi)\) then \(\varphi \in FL(\phi)\)
\((f_2)\) if \(\varphi \lor \psi \in FL(\phi)\) then \(\varphi \in FL(\phi) \text{ and } \psi \in FL(\phi)\)
\((f_3)\) if \(X\varphi \in FL(\phi)\) then \(\varphi \in FL(\phi)\)
\((f_4)\) if \(G\varphi \in FL(\phi)\) then \(\varphi \in FL(\phi)\)
\((f_5)\) if \(\varphi \lor \psi \in FL(\phi)\) then \(\varphi \in FL(\phi) \text{ and } \psi \in FL(\phi)\)
\((f_6)\) if \([\alpha]\varphi \in FL(\phi)\) then \(\varphi \in FL(\phi)\)
\((f_7)\) if \([\alpha \lor \beta]\varphi \in FL(\phi)\) then \([\alpha]\varphi \in FL(\phi) \text{ and } [\beta]\varphi \in FL(\phi)\)
\((f_8)\) if \([\alpha; \beta]\varphi \in FL(\phi)\) then \([\alpha][\beta]\varphi \in FL(\phi)\)
\((f_9)\) if \([\alpha^*]\varphi \in FL(\phi)\) then \([\alpha][\alpha^*]\varphi \in FL(\phi)\)
\((f_{10})\) if \([\alpha^*]\varphi \in FL(\phi)\) then \(\alpha^* \varphi \in FL(\phi)\)
\((f_{11})\) if \(\varphi \rightarrow \alpha/\beta \in FL(\phi)\) then \(\varphi \in FL(\phi)\)

Given a formula \(\varphi\) of \(L_A\) and a nonstandard model \(M = \langle W, r, l, fj, pr\rangle\), we define a new model \(M/FL(\varphi) = \langle W/FL(\varphi), r/FL(\varphi), l/FL(\varphi), fj, pr\rangle\), called the filtration of \(M\) by \(FL(\varphi)\), as follows. We define a binary relation \(\equiv\) on states of \(M\) by:

\[
\forall \psi \in FL(\varphi), (u \equiv v) \iff (u \in l(\psi) \iff v \in l(\psi))
\]

It is easy to see that the relation \(\equiv\) is reflexive, symmetric and transitive, and so an equivalence relation. Let

\[
[u] \overset{\text{def}}{=} \{v \mid u \equiv v\}
\]

\(W/FL(\varphi) \overset{\text{def}}{=} \{[u] \mid u \in W\}\)

\(l/FL(\varphi)(p) \overset{\text{def}}{=} \{[u] \mid u \in l(p)\}, p \text{ an atomic proposition}\)

\(r/FL(\varphi)(a) \overset{\text{def}}{=} \{([u], [v]) \mid (u, v) \in r(a)\}, a \text{ an atomic action}\)

The maps \(l/FL(\varphi)\) and \(r/FL(\varphi)\) are inductively extended to compound propositions and actions.

**Lemma 4.5.** Filtration for Nonstandard Models

*Let \(M\) be a nonstandard model and let \(u, v\) be states of \(M\).*

(i) For all \(\psi \in FL(\varphi), u \in l(\psi)\) iff \([u] \in l/FL(\varphi)(\psi)\)

(ii) For all \([\alpha]\psi \in FL(\varphi),\)

(a) if \((u, v) \in r(\alpha)\) then \([u], [v]) \in r/FL(\varphi)(\alpha)\)

b if \(([u], [v]) \in r/FL(\varphi)(\alpha)\) and \(u \in l([\alpha]\psi),\) then \(v \in l(\psi)\).
Proof. The proof can be found in [43]. □

**Theorem 4.5.** Small Model Theorem

Let $\varphi$ be a satisfiable formula of $L_A$. Then $\varphi$ is satisfied in a finite standard model with no more than $2^{|\varphi|}$ states.

Proof. The proof of this theorem is similar to one in [43] (p. 199). □

The above proof shows that a finite standard model of the theorem 4.5 is $M/FL(\varphi)$.

**Theorem 4.6.** Completeness

$\forall \varphi, \models \varphi \Rightarrow \vdash \varphi$

Proof. We want to show that $\not\models \varphi \Rightarrow \not\models \varphi$, i.e., $\neg \varphi$ has a model.

If $\not\models \varphi$, then $\neg \varphi$ is consistent; then by lemma 4.1(2), $\neg \varphi$ is contained in a maximal consistent set $u$, which is a state of the nonstandard model $M_n = \langle W_n, r_n, l_n, f_j, pr \rangle$ constructed above ($\neg \varphi \in u$). That implies that $u \in l_n(\neg \varphi)$. $\varphi \in FL(\varphi)$ implies that $\neg \varphi \in FL(\varphi)$, by the definition of Fisher-Ladner closure ($f_1$). $\neg \varphi \in FL(\varphi)$ and $u \in l_n(\neg \varphi)$ imply by the filtration lemma 4.5 (i) that $[u] \in l_n/FL(\varphi)(\neg \varphi)$. This implies that $\neg \varphi$ is satisfied at the state $[u]$ in the finite model $M_n/FL(\varphi)$, which is a standard model by definition. Thus, if $\neg \varphi$ is consistent, then there exist a standard model $M_n/FL(\varphi)$ and a state $[u]$ such that $M_n/FL(\varphi), [u] \models \neg \varphi$. Therefore $\neg \varphi$ has a model. □

### 4.7 Related Work

Our work is related to the work of Prisacariu and Schneider [74, 75]. They use deontic logic and propositional dynamic logic to write electronic contracts. Their approach is specific to electronic contracts and still have technical difficulties in the underlying semantics due to properties the logic needs to capture. Our approach is more general (i.e., it is not specific to electronic contracts) in the sense it can be applied to many kind of systems, since aspect-oriented technology is intended to model software systems. Dignum and Kuiper [22] combine dynamic deontic logic and temporal logic to specify deadlines. Each agent has an agenda (a set of deontic temporal constraints), which specify what the agent should do. They adopt the *ought-to-be* approach while our approach is based on *ought-to do*. Moreover, they do not give inference rules for their logic. Wiels [110] used a combination of dynamic logic and linear temporal logic to formally specify and verify object-oriented systems. Deontic modalities in our approach are useful and necessary to specify our prevention policy for aspect-oriented systems, since deontic logic
is a logic for ideal and actual behavior. The usefulness of deontic logic for modeling the
behavioural aspects of systems has been recently pointed out by several researchers (in
[30]). The deontic modalities are becoming increasingly of interest in computer science,
with proposed applications including intelligent legal information systems, computer
security, software engineering, database integrity constraints, and agent-oriented pro-
gramming [104]. Others have shown that such a combination (LTL + Dynamic Logic
+ Deontic Logic) is adequate for systems specification [42, 48] because it has the ad-
vantage to distinguish between description and prescription of behavior. Fiadeiro and
Maibaum [29, 30] adopt a deontic action logic to formally specify and verify object-
oriented systems. Their logic is a combination of positional (modal) operators to define
the effects of actions, dynamic logic, and deontic logic to define the behavioral aspects.
Our logic is for the specification and the verification of aspect-oriented systems. Our
treatment of permissions and obligations is quite similar to the dynamic logic of permis-
sion presented by van der Meyden in [104]. But our action constructs go beyond van
der Meyden’s logic. (These constructs seem promising to allow interactive construction
or verification of protocols.) Also, our logic has violation atoms, which is not the case
in van der Meyden’s logic.

4.8 Classical Paradoxes

Deontic logic may be described as a logic of prohibitions, permissions and obliga-
tions. The paradoxes in this field are logical expressions that are valid in a (or even most)
well-known logical system(s) for deontic reasoning, but which are counterintuitive in the
common sense reading [59]. In this section, we briefly present classical paradoxes that
arise in deontic logic. Our goal is to avoid as many deontic logic paradoxes as possi-
ble. Aspect-oriented technology can be used to model any kind of system in software
engineering. We use the logic $L_A$ to specify the behavior and the societal life of the
components of a system. Therefore, it is reasonable to see that deontic paradoxes (which
are often quoted in the literature) are not harmful for the logic $L_A$. It has been shown
in the literature [58] that the ought-to-do approach allows to avoid or to resolve some
classical paradoxes of the standard deontic logic.

4.8.1 Ross’s Paradox

Informally, it is expressed as:
1. It is obligatory that one mails the letter.

2. It is obligatory that one mails the letter or one burns the letter.

In Standard Deontic Logic (SDL) these are expressed as:

1. \( O(\alpha) \)

2. \( O(\alpha \lor \beta) \)

The problem is that in SDL one can infer that \( O(\alpha) \Rightarrow O(\alpha \lor \beta) \).

Meyer [58] solves this paradox by defining an obligation operator \( O' \) such that 
\[ O'\alpha \equiv O\alpha \land P\alpha \land \neg [\alpha] \neg V \equiv (\alpha) \neg V \land [\alpha] \neg V \] i.e., it is possible to do \( \alpha \) without getting into trouble and every way of doing \( \alpha \) is allowed. Thus by adopting this solution, Ross’s paradox is not harmful for our logic (since, \( O'(\alpha) \Rightarrow O'(\alpha \lor \beta) \) no longer holds).

### 4.8.2 Chisholm’s Paradox

In natural language:

1. John ought to go to the party.

2. If John goes to the party then he ought to tell them he is coming.

3. If John does not go to the party then he ought not to tell them he is coming.

4. John does not go to the party.

In SDL this is:

1. \( O(\alpha) \)

2. \( O(\alpha \rightarrow \beta) \)

3. \( \neg \alpha \rightarrow O(\neg \beta) \)

4. \( \neg \alpha \)
The problem is that in SDL one can infer $O(\beta) \land O(\neg \beta)$ which is due to statement (2). However, in our logic, this paradox is not harmful since, the points 1 to 4 can be written without any problem by the following formula: $O\alpha \land [\alpha]O\beta \land [\neg \alpha]O(\neg \beta)$. The problem in SDL was that one may infer both $O(\beta)$ and $O(\neg \beta)$ holding in the same world. This is not our case because $O(\beta)$ holds only after doing action $\alpha$, where $O(\neg \beta)$ holds only after doing the contradictory action $\alpha$. In the model of the above representation we can not have in the same world both $O(\beta)$ and $O(\neg \beta)$.

### 4.8.3 The Free Choice Permission Paradox

Informally, we have:

1. You may either sleep on the sofa or sleep on the bed.
2. You may sleep on the sofa and you may sleep on the bed.

In SDL this is:

1. $P(\alpha \lor \beta)$
2. $P(\alpha) \land P(\beta)$

The natural intuition tells that $P(\alpha \lor \beta) \Rightarrow P(\alpha) \land P(\beta)$. In SDL this would lead to $P(\alpha) \Rightarrow P(\alpha) \lor P(\beta)$ which is $P(\alpha) \Rightarrow P(\alpha) \land P(\beta)$ (because $P(\alpha \lor \beta) \equiv\_d P(\alpha) \lor P(\beta)$ (see theorem 4.2) and $P(\alpha \lor \beta) \Rightarrow P(\alpha) \land P(\beta)$), so $P(\alpha) \Rightarrow P(\beta)$.

As an example: If one is permitted something, then one is permitted anything. Meyer resolved this paradox by the free choice operator $P_F$: $P_F \alpha \equiv P\alpha \land [\alpha]\neg V$. $P_F$ has the following property: $P_F(\alpha \lor \beta) \equiv (P_F \alpha \land [\beta]\neg V) \lor (P_F \beta \land [\alpha]\neg V)$. We don’t have $P_F \alpha \Rightarrow P_F(\alpha \lor \beta)$.

### 4.8.4 The Gentle Murderer Paradox

In natural language:

1. It is obligatory that John not kill his mother.
2. If John does kill his mother, then it is obligatory that John kill her gently.
3. John does kill his mother.

In SDL this is:
1. $O(\neg\alpha)\\n2. \alpha \rightarrow O(\beta)\\n3. \alpha\\nThe problem is that when adding a natural inference like $\beta \rightarrow \alpha$ then in SDL one can infer that $O(\alpha)$. However, by posing $\alpha_1 = \neg\alpha$ and $\beta_1 = \neg\beta$ the above set can be represented in our logic as:\n\n1. $O(\alpha_1)\\n2. [\alpha_1]O\beta_1\\nwhich are the points 1 and 3 of Chisholm’ set. So, the Gentle Murderer Paradox is eliminated in our logic as in case of Chisholm’ paradox.

No of the paradoxes presented above is harmful for the logic $L_A$.

4.9 Conclusion

In this chapter, we described a logic $L_A$ to be used to specify the prevention policy, but also the properties describing the behaviors of aspect and class components of an aspect-oriented system. The prevention policy describes the societal life of these components. This logic is a combination of temporal, dynamic, and deontic logics. We showed that the proof system of $L_A$ is sound and complete. The next chapter covers the syntax and the semantics of our approach.
CHAPTER 5

SYNTAXES AND SEMANTICS OF OUR APPROACH

5.1 Introduction

In Aspect-Oriented Technology, system components can be aspect or class components, which use data structures. Aspect components, class components, and abstract data types are described by aspect, class, and abstract data type module specifications, respectively. As in Wiels’ approach, there are three levels of a system description in our approach:

1. a system is described by modules that are interconnected by morphisms and on which composition operations can be performed;

2. a module is composed of four specifications linked by specification morphisms;

3. each specification consists of a vocabulary part and a set of formulae describing the behavior and the constraints of this specification. The formulae are written with the logic $\mathcal{L}_A$ (see section 4).

We equip module specifications with a prevention policy consisting of a set of properties expressing the societal life of the components. These properties are expressed by our logic in the form of social norms forbidding the undesirable interactions and behaviors that a component could have. Any violation of these norms will be sanctioned. This chapter describes the concrete and the abstract syntaxes, and the semantics of our approach.

5.2 Specifications

This section defines the concrete syntax and the abstract syntax of an algebraic specification. We define the different kinds of specification.

5.2.1 Concrete Syntax of Specifications

The components of a system can be aspects or classes. Thus, we have to consider aspect and class specifications. These components use data types; this led us to consider
abstract data type (ADT) specification. The definition of these kinds of specification is based on work of Ehrig and Mahr [24]. Ehrig and Mahr’ work is an equational theory while our work is a logical theory.

5.2.1.1 Specification of an Abstract Data Type

The two basic notions in computer science are the notions of operations and data. Data structures and data types are fundamental concepts of programming and specification of software systems. A set of data is called data domain. A collection of data domains and operations is called data type. An Abstract Data Type (ADT) is a class of data types which is closed under renaming, items and operations and hence independent of any specific programming language or representation [24]. At the design level (design of algorithms and software systems), it is not interesting to consider concrete representation on a data type. A set of operations is associated to each abstract data type, making it an algebra. We can thus define the specification of this algebra. Specification of a abstract data type concerns the definition of classes of algebras, while specification of a data type concerns a single algebra. Figure 5.1 defines the pattern of an ADT specification. The pattern of an ADT specification is composed of:

1. a header with reserved word SPEC_AD T followed by the specification name.

2. a field Sorts which lists the imported sorts plus the sort corresponding to the current ADT. For example, if the name of the ADT specification is Bool, the sort of Bool is noted by bool.

3. a field Subsorts lists the subsort realtions of the specification under consideration. In order to model sort inclusions and coercions, we equip the specification with an partial order relation $\leq$ over the set of sorts. Thus, overloading is supported and operations are allowed to be sub-sort polymorphic. Hence, we increase the expressiveness of our algebraic specification.

4. a field Actions which contains the operation signatures of the ADT. It contains also the constructors. Actions could be partial or total operations. We choose this term Action to be conform to our logic $L_A$.

5. a field Axioms which defines the semantics of the operations by defining a set of axioms that characterize the behavior of the ADT. These axioms written with
**SPEC_ADT** Specification Name

**Sorts** (list of sorts name)

**Subsorts** (list of subsorts relations)

**Actions** (operation signatures setting out the names and the sorts of the parameters of the operations and the return sort)

**Axioms** (operations properties)

**Theorems** (possible theorems)

**End** Specification Name

---

Figure 5.1: Pattern of an ADT Specification

our logic $L_A$ could be formulae or positive conditional equations of the form: $Conditions \Rightarrow equation$.

6. a **Theorems** field contains possible formulas which are assumed to be true. These formulas are generally consequences derived from the axioms.

7. the specification ends with the reserved word **END** followed by the specification name.

Thus, our specification pattern defines a class of partial order-sorted algebras (because of subsort relation).

**Example 5.1.** A string is a sequence of data items from some given data domain. It is a word $a_1 \ldots a_n$ of length $n \geq 0$ and $a_1, \ldots, a_n$ belongs to some set or alphabet $A$. For $n = 0$, it is the empty string. This example defines the specification ADT String for the abstract data type STRING. Let $A$ be $A = \{a_1, \ldots, a_n\}$ with constants $a_1, \ldots, a_n$. The set of all strings is the domain $A^*$, i.e., the set of words $b_1 \ldots b_m, m \geq 0, b_i \in A$ for $i \in \{1, \ldots, m\}$ of length $n \geq 0$. The operations that we consider on strings (which includes the empty constant $\lambda$) are:

- make: $A \rightarrow A^*$ such that $\text{return}(a) = a$; it makes a string of length 1 for each element in $A$.

- concat: $A^* \times A^* \rightarrow A^*$ such that $\text{concat}(u, v) = uv$; it makes the concatenation of strings

...
- ladd: $A \times A^* \rightarrow A^*$ such that $\text{ladd}(a, u) = au$; it adds an element of the alphabet on the left side of the string.
- radd: $A^* \times A \rightarrow A^*$ such that $\text{radd}(u, a) = ua$; it adds an element of the alphabet on the right side of the string.

The algebraic model of the abstract data type String is:

$\text{STRING} = \langle A, A^*, a_1, \ldots, a_n, \text{empty}, \text{make}, \text{concat}, \text{ladd}, \text{radd} \rangle$.

```
SPEC_ADT String
    Sorts: alphabet, string
    Actions:
        a_1, \ldots, a_n :\rightarrow\ alphabet
        EMPTY :\rightarrow\ alphabet
        MAKE :alphabet \rightarrow\ string
        LADD :alphabet string \rightarrow\ string
        RADD :string alphabet \rightarrow\ string
    Axioms:
        a: alphabet
        s, s_1, s_2, s_3: string
        G(\forall s, \text{CONCAT}(s, \text{EMPTY}) = s)
        G(\forall s_1, s_2, s_3, \text{CONCAT}(\text{CONCAT}(s_1, s_2), s_3) =
            \text{CONCAT}(s_1, \text{CONCAT}(s_2, s_3)))
        G(\forall a, s, \text{LADD}(a, s) = \text{CONCAT}(\text{MAKE}(a), s))
        G(\forall a, s, \text{RADD}(s, a) = \text{CONCAT}(s, \text{MAKE}(a)))
END String
```

Figure 5.2: Specification of ADT String

The figure 5.2 presents the specification String of the abstract data type STRING.

5.2.1.2 Specification of a Class

Although, originally, algebraic specification was intended as technique for the description of data type, it soon grew into formal specification technique aiming to cover the whole specification phase within the software development process [26]. Large systems are usually decomposed into sub-systems which are developed independently. Sub-systems are often defined as a set of abstract data types or object classes. A class is used to model the behavior of objects. It consists of a number of operations, especially
functions. Each of the operations has a signature which describes the types of its arguments and the type of the result. Classes are seen as templates of objects by describing the common aspects of these objects. Each object belongs to a class that specifies the behavior and attributes of its instances; objects collaborate to satisfy the concerns of the system (object in category context is more general than object as an instance of a class in object-oriented context). The core parts of a class describe which attributes can be observed, which actions can be performed, how attributes change when actions happen, under which conditions actions may happen, under which conditions actions are obliged to happen, etc. We are modeling computer programs as algebras. A specification defines a set of algebras (computer programs). Thus, classes can be described by algebraic specification technique. By doing this, algebraic specifications could be rendered more suitable for large applications on the one hand, and computer programs could be supported by specifications with well-defined semantics on the other hand. Each object has a state. Thus, we have a notion of state in our algebraic specification. There are many other approaches (such as [13]) that applied algebraic technique to object-oriented technology; they differ from one to another by the formalism used to describe the behavior of the components. The pattern of a Class specification is defined as follow: The pattern of a \textit{CLASS specification} is composed of:

1. a header with reserved word \textbf{SPEC\_CLASS} followed by the specification name.

2. a field \textbf{Sorts} which lists the imported sorts plus the sort corresponding to the current Class. For example, if the name of the CLASS specification is Buffer, the sort of the Class BUFFER (algebra) is noted by buffer.

3. a field \textbf{Subsorts} lists the subsort relations of the class specification under consideration. In order to model sort inclusions and coercions, we equip the specification with an partial order relation $\leq$ on the set of sorts. Thus, overloading is supported and operations are allowed to be sub-sort polymorphic.

4. a field \textbf{States} gives the list of the attribute signatures of the class instances. These attributes determine at a given time the state of an class instance or object. The template of an attribute signature is at: $s_1 s_2 \ldots s_n \rightarrow s$ with $n \geq 0$; at is the name of the attribute; $s_1, s_2, \ldots, s_n$ denote optional parameter sorts while $s$ denotes the range sort of the attribute. In programming terms, an attribute symbol which has no arguments corresponds to a program variable, whereas those with arguments can be associated with more complex data structures such as arrays.
5. the field **Initial** gives the initial state of an object or default valuers of the attributes.

6. a field **Actions** which contains the operation signatures of the Class. It contains also the constructors. Actions could be partial or total operations.

7. a field **Axioms** which defines the semantics of the operations by defining a set of axioms that characterize the behavior of the Class. These axioms written with our logic $L_A$ could be formulae or positive conditional equations of the form: $\text{Conditions } \Rightarrow \text{ equation}$.

8. a field **Prescription Axioms** which defines properties describing what the system should do, or proscribing the violation of desired properties, i.e., undesirable interactions.

9. a **Theorems** field contains possible formulae that are assumed to be true. These formulae are generally consequences derived from the axioms.

10. the specification ends with the reserved word **END** followed by the specification name.
Thus, our Class specification pattern defines a class of partial order-sorted algebras.

Example 5.2. We consider a simplistic example that has been taken from [101]. This example shown in figure 5.4 is composed of one class, Invoice, and two aspects, Log and Verification. It is represented in a UML notation. The class Invoice has two methods: (1) Tax(double) which calculates and adds a tax to an amount; (2) Interest(double) which calculates interests and adds this to an amount, in the case a buyer buys by credit option. The aspect Log prints messages and stores these into a file, and manages the applicable tax. The aspect Verification validates the inputs data. Each tuple of a table is associated with a pointcut. The columns contain elements of an aspect-class connection. One of our goal is to characterize in Ehrig and Marh [25] module interconnection formalism these kinds of relationship. This characterization will allow us to calculate the augmented modules (modules obtained after weaving). Figure 5.5 gives algebraic specification of the class Invoice.

Figure 5.4: Societal life of aspects and classes
**SPEC_CLASS** InvoiceS

Sorts: double, invoice, int, bool

States:
- amount: double → double

Actions:
- Tax: double → double
- Interest: double → double

Axioms
- amount: double
- ∀ amount,
  \[
  G([\text{Tax}(\text{amount})](\text{Tax}(\text{amount}) > \text{amount}))
  \]

End InvoiceS

**Figure 5.5: Specification of Invoice Class**

The specification InvoiceS contains one attribute and amount, two actions (methods) Tax and Interest, and one axiom \( G([\text{Tax}(\text{amount})](\text{Tax}(\text{amount}) > \text{amount})) \) that means: after all execution of the method Tax(amount), the property (Tax(amount) > amount) holds.

### 5.2.1.3 Specification of an Aspect

AOP is quantification. Quantification means that programs can include quantified statements (i.e. statements that apply to more than one place) of the form "In programs P, whenever condition C arises, perform action A". Aspect is a class-like constructs. Like classes, aspects can have attributes (states) and methods; in addition they encapsulate the remaining components of quantified statements: conditions are specified by point-cut expressions and actions are specified by method like constructs called advice [96] (see section 3.1). A pointcut is a predicate that matches join points. More precisely, a pointcut is a relationship from Join Point → boolean, where the domain of the relationship is all possible join points [11]. Thus, we specify an aspect by using algebraic specification technique. The pattern of an Aspect specification is defined as follow: The pattern of an *ASPECT specification* is composed of:

1. a header with reserved word **SPEC_ASPECT** followed by the specification name.

2. a field **Sorts** which lists the imported sorts plus the sort corresponding to the current Aspect. For example, if the name of the ASPECT specification
**SPEC_ASPECT** Specification Name

**Sorts**  
(list of sorts name)

**Subsorts**  
(list of subsorts relations)

**States**  
(list of attribute signatures of aspect instance)

**Initial**  
(initial state)

**Pointcuts**  
(list of pointcut signatures)

**Actions**  
(operation signatures setting out the names and the sorts of the parameters of the operations and the return sort)

**Pointcut Axioms**  
(pointcuts description)

**Advice Axioms**  
(advices description)

**Axioms**  
(operations properties)

**Prescription Axioms**  
(societal life properties)

**Theorems**  
(possible theorems)

**End Specification Name**

---

Figure 5.6: Pattern of an ASPECT Specification
is GoodAspect, the sort of the aspect GOODASPECT (algebra) is noted by goodAspect.

3. a field Subsorts lists the subsort relations of the aspect specification under consideration. In order to model sort inclusions and coercions, we equip the specification with an order relation on the set of sorts. Thus, overloading is supported and operations are allowed to be sub-sort polymorphic.

4. a field States gives the list of the attribute signatures of the aspect instance. These attributes determine at a given time the state of an aspect instance. The template of an attribute signature is at: $s_1 s_2 \ldots s_n \rightarrow s$ with $n \geq 0$; at is the name of the attribute; $s_1$, $s_2$, \ldots, $s_n$ denote optional parameter sorts while $s$ denotes the range sort of the attribute.

5. the field Initial gives the initial state of an aspect instance or default values of the attributes.

6. Pointcut field lists the pointcut signatures inside the aspect. A pointcut signature is comprised of its name, the sorts (types) of its optional parameter (source, target, and arguments of the joinpoints), and the result sort bool (boolean sort). The template of a pointcut signature is pc : $s_1 s_2 \ldots s_n \rightarrow \text{bool}$ with $n \geq 0$. For example, the signature of the pointcut defined in the figure 3.1 is:

setXYValue: Maintain mygraphic.Point int int -> bool

7. a field Actions which contains the operation signatures of the aspect; this includes the advice operation signatures (but the advice operations can not be called by the class components). It contains also the constructors. Actions could be partial or total operations.

8. Pointcut Axioms define the joinpoints contained in the pointcut expressions. In other words, it describes the selection action of the pointcuts.

9. Advice Axioms field gives the advice descriptions by using the logic $\mathcal{L}_A$.

10. a field Axioms which defines the semantics of the other operations by defining a set of axioms that characterize the behavior of the aspect. These axioms written with our logic $\mathcal{L}_A$ could be positive conditional equations of the form: $Conditions \Rightarrow equation.$
11. a field **Prescription Axioms** which defines properties describing what the system should do, or proscribing the violation of desired properties. These axioms are defined with the logic $L_A$. A prescription axiom contains at least one deontic operator.

12. a **Theorems** field contains possible formulae which are assumed to be true. These formulas are generally consequences derived from the axioms.

13. the specification ends with the reserved word **END** followed by the specification name.

Thus, our Aspect specification pattern defines a class of partial order-sorted algebras.

**Example 5.3.** Consider the example in the section 5.4. Examples of aspect specifications are given in figures 5.7 and 5.8. The specification of the aspect Verification contains among other, two pointcuts verifPt1 (it is without parameter), verifPt2: double (it has a parameter which is the argument of the joinpoint execution in.Interest(am), i.e., the pointcut verifPt2 will verify that am is of type double). pr(call in.Tax(am)) is the atomic proposition "the joinpoint call in.Tax(am) is reached". The advice axiom $(B_e, pr(call in.Tax(am)))(\text{condition1 } \rightarrow \text{println}(m_1)/\text{println}(m_2))$ means that immediately before the joinpoint call in.Tax(am) is reached, if condition1 holds then execute the method println($m_1$) followed by the execution of in.Tax(am), else execute the method println($m_2$) followed by the execution of in.Tax(am). The prescription axiom $(B_e, pr(call in.Tax(am)))(\text{I}(\text{println}(am) ; \text{condition1 } \rightarrow \text{println}(m_1)/\text{println}(m_2)))$ shows the order in which the two aspects Verification and Log should be executed in the class Invoice, regarding the two crosscutting methods println() of the aspect Log and println() of the aspect Verification, at the same joinpoint call in.Tax(am). It says that condition1 $\rightarrow$ println($m_1$)/println($m_2$) should be executed before the execution of println(am), at the joinpoint call in.Tax(am) (immediately before). This prescription axiom allows to avoid a conflicting situation that could be arisen; it is an example that prevents an undesirable interaction. The specification of the aspect Log is designed in the same optic.
### SPEC_ASPECT VerificationS

**Sorts:** double, string, invoice, verification, bool

**State:** amount: double $\rightarrow$ double

**Pointcuts**

- `verifPt1`: bool
- `verifPt2`: double $\rightarrow$ bool

**Actions**

- `println`: string
- `println`: double

**Pointcut Axioms:**

- `in`: invoice
- `verifPt1`: `pr(call in.Tax(amount))`
- `forall` `in`,
  - `verifPt2(amount)`: `pr(execution in.Interest(amount) 
  && args(amount))`

**Advice Axioms:**

- `in`: invoice, $m_1$, $m_2$: string
- `forall` `in`, $m_1$, $m_2$,
  - $(B_{pr}(call in.Tax(amount)))(condition1 \rightarrow println(m_1)/println(m_2))$
- `forall` `in`, $m_1$, $m_2$,
  - $(B_{pr}(execution in.Interest(amount) && args(amount)))$
  - $(condition2 \rightarrow println(m_1)/println(m_2))$

**Prescription Axioms:**

- `amount`: double, $m_1$, $m_2$: string, `in`: invoice
- `forall` $m_1$, $m_2$ in,
  - $(B_{pr}(call in.Tax(amount)))(I(println(amount);$
  - `condition1 \rightarrow println(m_1)/println(m_2))$

End VerificationS

---

**Figure 5.7:** Verification Aspect Specification
SPEF ASPECT LogS

Sorts: double, string, invoice, file, log, bool

States:

  tax: double → double
  amount: double → double

Pointcuts

  logPt1: bool
  logPt2: double → bool
  logPt3: bool

Actions

  println: string
  println: double
  ×: double double → double
  store: double file → file

Pointcut Axioms:

  in: invoice
  logPt1: pr(call in.Tax(amount))
  ∀ in, logPt2(amount): pr(call in.Tax(amount))
    && args(amount)

  logPt3: pr(call *.*)

Advice Axioms:

  in: invoice, m₁, m₂: string
  ∀ in,
    (B₁pr(call in.Tax(amount)))println(amount))
    (A₈pr(call in.Tax(amount) && args(amount)))
    (amount = proceed(amount); return(amount × tax))
    ∀ m₂, (A₈pr(call *.*))println(m₁)

Axioms:

  f: file
  ∀ f, [store(amount, f)](amount ∈ f)

Prescription Axioms:

  m₁, m₂: string, in: invoice
  ∀ m₁, m₂, in,
    (B₁pr(call in.Tax(amount))) println(amount);
    condition1 → println(m₁)/println(m₂))

End LogS

Figure 5.8: Log ASPECT Specification
Remark 5.1. It is worth to note that:

1. The pattern of an ADT specification is defined as the pattern of an aspect (Figure 5.6) without the fields: **States, Initial, Pointcuts, Pointcut Axioms, Advice Axioms, Prescription Axioms**, and in which the header is replaced by the reserved word **SPEC_ADT**.

2. The pattern of a Class specification is defined as the pattern of an aspect (Figure 5.6) with the following fields excluded: **Pointcuts, Pointcut Axioms, Advice Axioms**, and in which the header is replaced by the reserved word **SPEC_CLASS**. A difference between a class and an ADT is that an instance of a class can have states while an ADT does not. Also, an ADT defines data types of a system and hence does not need the **Prescription Axioms** field.

### 5.2.2 Abstract Syntax of Specifications

In general, an algebraic specification consists of three parts:

1. A signature, consisting of sort names and functions symbols, from which one can build data terms. Each function symbol $f$ is of the form (signature or interface) $f : s_1 s_2 \ldots s_n \rightarrow s$ for some $n \geq 0$, where $s_1, s_2, \ldots s_n, s$ are sort names; $f$ is said to have arity $s_1, s_2, \ldots s_n$, result sort (or co-arity) $s$, and rank (or declaration) $(s_1 s_2 \ldots s_n, s)$. This means that if $d_1, d_2, \ldots d_n$ are data terms of sorts $s_1, s_2, \ldots s_n$ respectively, then $f(d_1, d_2, \ldots d_n)$ is a data term of sort $s$. If $n = 0$, that is $f : \rightarrow s$, we use the abbreviation $f : s$ and $f$ is a constant; $f : s$ can be written $f : \lambda \rightarrow s$ where $\lambda$ is the empty string of sorts. If $S$ is a set of sorts, then we note $S^*$ the set of all strings of elements of $S$ ($S^*$ is the free monoid over $S$). If $\omega = s_1 s_2 \ldots s_n$, $\omega \in S^*$, we can note $f : \omega \rightarrow s$. The set of all function symbols $f : \omega \rightarrow s$ with $\omega \in S^*$ is noted $\Omega_{\omega,s}$.

2. For each sort $s$ a countably infinite set of data variables of sort $s$.

3. A set of axioms, i.e., equations $d = e$ between data terms (possibly containing data variables) of same sort, which induces an equality relation on data terms.

We define our signature as an ordered sorted signature [39], because of the benefits (polymorphism, overloading, error detection, partial operation, etc.). A signature defines the specific vocabulary symbols that are relevant for the description of the specification.
A signature or interface defines the boundaries (limits) of the specification representing a component. The components of a system will interact via this interface, enforcing the principle of the encapsulation. When we reason about a component, we must use only the vocabulary (the language or the signature) of that component. We define abstract syntax for each of three kinds of specification presented in above sections. The abstract syntax definition is need to manipulate specifications, for instance to define semantics of specifications and module specifications.

5.2.2.1 ADT Specification

**Definition 5.1** (ADT signature). An **ADT signature** is a tuple \( \langle S, \leq, AC, \tau^A \rangle \) where \( S \) is a finite set of sorts, \( \leq \) is a partial order defined over \( S \), \( AC = \bigcup_{\omega \in S^*, s \in S} (AC_{\omega,s}) \) a finite set of action signatures where \( \{AC_{\omega,s}\}_{\omega \in S^*, s \in S} \) is a \( S^* \times S \)-sorted set of pairwise disjoint subsets of \( AC \), and \( \tau^A \) is an action profile function s.t. \( \tau^A : AC \to S^* \times S \) and if \( f : s_1 \ s_2 \ldots \ s_n \to s \in AC \), \( \tau^A(f : s_1 \ s_2 \ldots \ s_n \to s) = \langle s_1 \ s_2 \ldots \ s_n, s \rangle \). \( \langle S, \leq \rangle \) is called a poset.

**Remark 5.2.** If \( \omega = s_1 \ s_2 \ldots \ s_n, \omega \in S^* \), we can note \( f : \omega \to s \) instead of \( f : s_1 \ s_2 \ldots \ s_n \to s \). The set of all action signatures \( f : \omega \to s \) with \( \omega \in S^* \) is noted \( AC_{\omega,s} \). The notation \( f : \omega \to s \in AC \) means that \( f : \omega \to s \in AC_{\omega,s} \). Note that, if we write \( f \in AC \) or \( f \in AC_{\omega,s} \) we mean the action signature \( f \), consisting of a name and a declaration.

**Remark 5.3.** If \( \sigma = s_1 s_2 \ldots s_n \in S^* \) and \( \sigma' = s'_1 s'_2 \ldots s'_n \in S^* \), \( \sigma \leq \sigma' \) means that \( s_1 \leq s'_1, s_2 \leq s'_2, \ldots, s_n \leq s'_n \).

**Definition 5.2. ADT specification.**

An **ADT specification** is a tuple \( \text{SPEC\_ADT} = \langle \Sigma, \text{VAR}, \text{AX} \rangle \) where \( \Sigma = \langle S, \leq, AC, \tau^A \rangle \) is an ADT signature, \( \text{VAR} = \bigcup_{s \in S} (\text{VAR}_s) \) is a finite set of variables where \( \{\text{VAR}_s\}_{s \in S} \) is a \( S \)-sorted set of pairwise disjoint subsets of \( \text{VAR} \), and \( \text{AX} \) is a set of axioms (defined on the set of data terms) that describe the behavior of the specification, and that are written in \( L_\text{A} \).

**Example 5.4.** An example of ADT specification.

The abstract syntax of the ADT specification String defined in the figure 5.2 is \( \text{String} = \langle \langle S, \leq, AC, \tau^A \rangle, \text{VAR}, \text{AX} \rangle \) such that:
\[ S = \{ \text{alphabet, string} \} \]
\[ \leq = \{ (\text{alphabet, alphabet}), (\text{string, string}) \} \]

AC = \{ a_1, \ldots, a_n : \rightarrow \text{alphabet}, \text{EMPTY} : \rightarrow \text{alphabet},
MAKE : \text{alphabet} \rightarrow \text{string}, \text{LADD} : \text{alphabet string} \rightarrow \text{string},
RADD : \text{string alphabet} \rightarrow \text{string} \} \]

\( \tau^A \) is the operation profile function associated.

VAR = \{ a : \text{alphabet, s, s}_1, s_2, s_3 : \text{string} \}

\[ \text{AX} = \{ G(\forall s, \text{CONCAT}(s, \text{EMPTY}) = s), \]
\[ G(\forall s_1, s_2, s_3) \]
\[ \text{CONCAT}(\text{CONCAT}(s_1, s_2), s_3) = \text{CONCAT}(s_1, \text{CONCAT}(s_2, s_3)), \]
\[ G(\forall a, s, \text{LADD}(a, s) = \text{CONCAT}(\text{AKE}(a), s)), \]
\[ G(\forall a, s \text{RADD}(s, a) = \text{CONCAT}(s, \text{MAKE}(a))) \} \]

### 5.2.2.2 Class Specification

Classes are described on the basis of conventional data type specifications. The basic difference between data values as instances of data types and objects as instances of object classes is that values are regarded as stateless items while objects are associated with an inherent notion of state. A Class specification consists of a signature (defining the specific vocabulary symbols that are relevant for the description of the class instances), and a collection of formulae of the language generated from the signature.

**Definition 5.3. Class signature.**

An **Class signature** is a tuple \( \langle S, \leq, \text{AC}, \text{ST}, \tau^A, \tau^S \rangle \) s.t. \( \langle S, \leq, \text{AC}, \tau^A \rangle \) satisfies the constraints of an ADT signature, \( \text{ST} = \bigcup_{\omega \in S^*, s \in S} (\text{ST}_{\omega, s}) \) is a finite set of attribute signatures where \( \{ \text{ST}_{\omega, s} \}_{\omega \in S^*, s \in S} \) is a \( S^* \times S \)-sorted set of pairwise disjoint subsets of \( \text{ST} \), and \( \tau^S \) is an attribute profile function s.t. \( \tau^S : \text{ST} \rightarrow S^* \times S \) and if \( f : s_1 s_2 \ldots s_n \rightarrow s \in \text{ST}, \tau^S(f) = \langle s_1 s_2 \ldots s_n, s \rangle \).

**Definition 5.4. Class specification.**

A **Class specification** is a tuple \( \text{SPEC}_{\text{CLASS}} = \langle \Sigma, \text{VAR, AX, PrAX} \rangle \) where \( \Sigma = \langle S, \leq, \text{AC, ST, } \tau^A, \tau^S \rangle \) is a Class signature, \( \text{VAR} = \bigcup_{s \in S} (\text{VAR}_s) \) is a finite set of vari-
ables where \( \{\text{VAR}_s\}_{s \in S} \) is a \( S \)-sorted set of pairwise disjoint subsets of \( \text{VAR} \), \( \text{AX} \) is a finite set of axioms (defined on the set of data terms) that describe the behavior of the class specification, and \( \text{PrAX} \) is a finite set of prescription axioms. The elements of \( \text{AX} \) and \( \text{PrAX} \) are written with \( L_A \).

**Example 5.5.** An example of Class specification.

The abstract syntax of the Class specification \( \text{InvoiceS} \) defined in the figure 5.5 is

\[ \text{InvoiceS} = \langle \langle S, \leq, AC, ST, \tau^A, \tau^S \rangle, \text{VAR}, \text{AX}, \text{PrAX} \rangle \] such that:

\[ S = \{ \text{double}, \text{invoice}, \text{int}, \text{bool} \} \]

\[ \leq = \{ (\text{double}, \text{double}), (\text{invoice}, \text{invoice}), (\text{int}, \text{int}), (\text{bool}, \text{bool}) \} \]

\[ AC = \{ \text{Tax} : \text{double} \rightarrow \text{double}, \text{Interest} : \text{double} \rightarrow \text{double} \} \]

\[ ST = \{ \text{amount} : \text{double} \rightarrow \text{double} \} \]

\( \tau^A \) is the associated operation profile function.

\( \tau^S \) is the associated attribute profile function.

\[ \text{VAR} = \emptyset \]

\[ \text{AX} = \{ G(\forall \text{amount}[\text{Tax(amount)}][\text{Tax(amount)}] > \text{amount}) \} \]

\[ \text{PrAX} = \emptyset \]

**5.2.2.3 Aspect Specification**

Like classes, aspects can have attributes (whose values at a given time determine their states), methods, pointcut expressions, and advices. Thus aspects are described on the basis of conventional data type specifications, with the notions of state, point-cut, and advice.

**Definition 5.5. Aspect signature.**

An Aspect signature is a tuple \( \langle S, \leq, AC, ST, PC, \tau^A, \tau^S, \tau^P \rangle \) s.t. \( \langle S, \leq, AC, ST, \tau^A, \tau^S \rangle \) satisfies the constraints of a class signature, \( PC = \bigcup_{o \in S^*} (PC_{o, \text{bool}}) \) is a finite set of pointcut signatures with bool the boolean sort and where \( \{PC_{o, b}\}_{o \in S^*} \) is a \( S^* \)-sorted set of pairwise disjoint subsets of \( PC \), and \( \tau^P \) is a pointcut profile function s.t. \( \tau^P : PC \rightarrow S^* \times \text{bool} \) and if \( np : s_1 \ s_2 \ldots s_n \rightarrow \text{bool} \in PC \), \( \tau^P(np) = \langle s_1 \ s_2 \ldots s_n, \text{bool} \rangle \).

**Definition 5.6. Aspect specification.**

An Aspect specification is a tuple \( \text{SPEC\_ASPECT} = \langle \Sigma, \text{VAR}, \text{AX}, \text{PAX}, \text{AAX}, \text{PrAX} \rangle \) where \( \Sigma = \langle S, \leq, AC, ST, PC, \tau^A, \tau^S, \tau^P \rangle \) is an Aspect signature, \( \text{VAR} = \bigcup_{s \in S} (\text{VAR}_s) \) is
a finite set of variables where \( \{ \text{VAR}_e \}_{e \in S} \) is a \( S \)-sorted set of pairwise disjoint subsets of \( \text{VAR} \), \( \text{AX} \) is a finite set of axioms (defined on the set of data terms), \( \text{PAX} \) is a finite set of point-cut axioms, \( \text{AAX} \) is a finite set of advice axioms, and \( \text{PrAX} \) is finite set of prescription axioms. The elements of \( \text{AX} \cup \text{PAX} \cup \text{AAX} \cup \text{PrAX} \) are written with the logic \( \mathcal{L}_A \), and describe the behavior and societal life of the aspect specification.

**Example 5.6.** An example of aspect specification.

The abstract syntax of the aspect specification Verification defined in the figure 5.7 is:

\[
\text{Verification} = \langle \langle S, \leq, \text{AC}, \text{ST}, \text{PC}, \tau^A, \tau^S, \tau^P \rangle, \text{VAR}, \text{AX}, \text{PAX}, \text{AAX}, \text{PrAX} \rangle
\]

such that:

- \( S = \{ \text{double}, \text{string}, \text{verification}, \text{bool} \} \)
- \( \leq = \{ (\text{double, double}), (\text{string, string}), (\text{invoice, invoice}), (\text{verification, verification}), (\text{bool, bool}) \} \)
- \( \text{AC} = \{ \text{println : string println : double} \} \)
- \( \text{ST} = \{ \text{amount: double }\rightarrow\text{ double} \} \)

\( \tau^A \) is the associated operation profile function.

\( \tau^S \) is the associated attribute profile function.

\( \tau^P \) is the associated pointcut profile function.

\[ \text{VAR} = \{ \text{in: invoice, m}_1, m_2 : \text{string} \} \]

\[ \text{AX} = \emptyset \]

\[ \text{PAX} = \{ \text{verifPt1: pr\langle \text{call in.Tax(amount)} \rangle}, \text{verifPt2(amount): pr\langle \text{execution in.Interest(amount) }\&\&\text{ args(amount)} \rangle} \} \]

\[ \text{AAX} = \{ (B_e\text{pr\langle \text{call in.Tax(am) }\rangle})(\text{condition1 }\rightarrow\text{ println(m}_1)/\text{println(m}_2)), (B_e\text{pr\langle \text{execution in.Interest(amount) }\&\&\text{ args(amount)} \rangle})(\text{condition2 }\rightarrow\text{ println(m}_1)/\text{println(m}_2)) \} \]

\[ \text{PrAX} = \{ (B_e\text{pr\langle \text{call in.Tax(amount) }\rangle})(\text{I\langle \text{println(amount) ;condition1 }\rightarrow\text{ println(m}_1)/\text{println(m}_2) \rangle}) \} \]
5.2.2.4 Terms

**Definition 5.7.** Terms definition.
A term is a well-formed expression built from the action, attribute and pointcut symbols, and variables (on elements of a signature Σ). "well formed" means that it respects the arities of all the symbols. Formally, terms are defined inductively:

- any constant c of sort s is a term of sort s;
- any variable x of sort s is a term of sort s;
- if \( t_1 \) is a term of sort \( s_1, \ldots, t_n \) a term of sort \( s_n \) and \( f : s_1 \ldots s_n \to s \) is an action symbol, then \( f(t_1, \ldots, t_n) \) is a term of sort s.
- if \( t_1 \) is a term of sort \( s_1, \ldots, t_n \) a term of sort \( s_n \) and \( f : s_1 \ldots s_n \to s \) is an attribute symbol, then \( f(t_1, \ldots, t_n) \) is a term of sort s.
- if \( t_1 \) is a term of sort \( s_1, \ldots, t_n \) a term of sort \( s_n \) and \( f : s_1 \ldots s_n \to \text{bool} \) is a pointcut symbol, then \( f(t_1, \ldots, t_n) \) is a term of sort bool.

The set of all terms over Σ and X (a S-sorted set of variables) is denoted \( T_Σ(X) \). The set of all terms over Σ and X with sort s is denoted \( (T_Σ(X))_s \); \( T_Σ(X) = ((T_Σ(X))_s)_{s \in S} \). A term is called a ground term if it contains no variables. The set of ground terms over Σ is denoted \( T_Σ \). If \( s \leq s' \), then \( (T_Σ(X))_s \subseteq (T_Σ(X))_{s'} \).

**Example 5.7.** The signature Σ of Boolean algebra consists of the function symbols \( \land, \lor, \neg, 0, 1 \). \( T_Σ(X) \) is the set of propositional formulas over \( \land, \lor, \neg, 0, 1 \). X is the set of propositional letters.

5.2.2.5 Well-Formed Formulae Relative to a Signature

Let Σ be a signature (it could be an ADT, Class or Aspect signature), X a S-sorted set of variables. An equation is a formal expression \( t_1 = t_2 \) consisting of two terms of the same sort separated by the equality symbol \( = \). A conditional equation is a formal expression \( t \to (t_1 = t_2) \) consisting of three terms where \( t_1, t_2 \) are of the same sort and \( t \) is of sort bool.

The set \( \mathcal{F} \) of the well-formed formulae relative to a \( Σ \) is defined as follows:

- \( \bot, \top \in \mathcal{F} \)
• an atomic proposition \( p \in \mathcal{F} \)

• \( \forall t_1, t_2 \in (T_{X}^{A}(X))_s, \ t_1 = t_2 \in \mathcal{F} \)

• \( \forall t_1, t_2 \in (T_{X}^{A}(X))_s, \ t \in (T_{X}^{A}(X))_{\text{bool}}, t \rightarrow (t_1 = t_2) \in \mathcal{F} \)

• A formula \( \phi \) of \( L_A \) is an element of \( \mathcal{F} \).

### 5.2.3 Specification Morphism

A **morphism** between two objects (in general) specifies the way these two objects interact with each other. The mean of this interaction or relationship is up to the user or system specifier. The focus that category theory puts on morphisms as structure-preserving mappings is paramount for software architectures because it is the morphisms that determine the nature of the interconnections that can be established between objects (system components) [28]. As structure-preserving mapping, a morphism between two objects is defined in such way we can consider one of them as a component of the second one. Specifications morphisms are defined by signature morphisms. As in [31], we are merely requiring that a signature morphism identifies the symbols in the target signature that are used to interpret or implement the symbols of the source signature. It renames the vocabulary of the source signature in the target signature. We define ADT specification morphism, Class specification morphism, Aspect specification morphism, and Aspect-Class specification morphism. In the following definitions, \( \sigma_i^* \) is the "mapcar of \( \sigma_i \)", i.e., an extension of \( \sigma_i \) to the free monoid \( S_i^* \) for \( i \in \{1, 2\} \).

#### 5.2.3.1 Signature Morphism

**Definition 5.8.** ADT signature morphism.

Given ADT signatures \( \Sigma_1 = \langle S_1, \leq_1, AC_1, \tau_1^A \rangle \) and \( \Sigma_2 = \langle S_2, \leq_2, AC_2, \tau_2^A \rangle \), an **ADT signature morphism** \( \sigma : \Sigma_1 \rightarrow \Sigma_2 \) consists of a pair of maps \( (\sigma_1, \sigma_2) \) where \( \sigma_1 : S_1 \rightarrow S_2 \) and \( \sigma_2 : AC_1 \rightarrow AC_2 \) such that:

- \( \forall f : s_1 s_2 \ldots s_n \rightarrow s \in AC_1, \ \sigma_2(f) : \sigma_1(s_1) \sigma_1(s_2) \ldots \sigma_1(s_n) \rightarrow \sigma_1(s) \in AC_2. \)

- \( \forall s_1, s_2 \in S_1, s_1 \leq_1 s_2 \Rightarrow \sigma_1(s_1) \leq_2 \sigma_1(s_2). \)
Remark 5.4. The first condition can also be expressed as follows: the diagram
\[
\begin{array}{c}
AC_1 \xrightarrow{\tau_1^A} S_1^* \times S_1 \\
\downarrow \sigma_2 \quad \downarrow \sigma_1^* \times \sigma_1 \\
AC_2 \xrightarrow{\tau_2^A} S_2^* \times S_2 
\end{array}
\]
commutes.

Definition 5.9. **Class signature morphism.**
Given Class signatures \( \Sigma_1 = \langle S_1, \leq_1, AC_1, ST_1, \tau_1^A, \tau_1^p \rangle \) and \( \Sigma_2 = \langle S_2, \leq_2, AC_2, ST_2, \tau_2^A, \tau_2^p \rangle \), a **Class signature morphism** \( \sigma : \Sigma_1 \to \Sigma_2 \) consists of a triple of maps \( (\sigma_1, \sigma_2, \sigma_3) \) where \( \sigma_1 : S_1 \to S_2 \), \( \sigma_2 : AC_1 \to AC_2 \), and \( \sigma_3 : ST_1 \to ST_2 \) such that:

- \( (\sigma_1, \sigma_2) : \langle S_1, \leq_1, AC_1, \tau_1^A \rangle \to \langle S_2, \leq_2, AC_2, \tau_2^A \rangle \) is an ADT signature morphism.

- \( \forall \text{at : } s_1 s_2 ... s_n \to s \in ST_1, \sigma_3(\text{at}) : \sigma_1(s_1) \sigma_1(s_2) ... \sigma_1(s_n) \to \sigma_1(s) \in ST_2 \).

Remark 5.5. The second condition can also be expressed as follows: the diagram
\[
\begin{array}{c}
ST_1 \xrightarrow{\tau_1^S} S_1^* \times S_1 \\
\downarrow \sigma_3 \quad \downarrow \sigma_1^* \times \sigma_1 \\
ST_2 \xrightarrow{\tau_2^S} S_2^* \times S_2 
\end{array}
\]
commutes.

Definition 5.10. **Aspect signature morphism.**
Given Aspect signatures \( \Sigma_1 = \langle S_1, \leq_1, AC_1, ST_1, PC_1, \tau_1^A, \tau_1^S, \tau_1^p \rangle \) and \( \Sigma_2 = \langle S_2, \leq_2, AC_2, ST_2, PC_2, \tau_2^A, \tau_2^S, \tau_2^p \rangle \), an **Aspect signature morphism** \( \sigma : \Sigma_1 \to \Sigma_2 \) consists of a 4-tuple of maps \( (\sigma_1, \sigma_2, \sigma_3, \sigma_4) \) where \( \sigma_1 : S_1 \to S_2 \) s.t. \( \sigma_1(\text{bool}) = \text{bool} \), \( \sigma_2 : AC_1 \to AC_2 \), \( \sigma_3 : ST_1 \to ST_2 \), and \( \sigma_4 : PC_1 \to PC_2 \) such that:

- \( (\sigma_1, \sigma_2, \sigma_3) : \langle S_1, \leq_1, AC_1, ST_1, \tau_1^A, \tau_1^S \rangle \to \langle S_2, \leq_2, AC_2, ST_2, \tau_2^A, \tau_2^S \rangle \) is a class signature morphism.
\[ \forall pc: s_1 s_2 \ldots s_n \rightarrow \text{bool} \in PC_1, \; \sigma_4(pc): \sigma_1(s_1) \sigma_1(s_2) \ldots \sigma_1(s_n) \rightarrow \sigma_1(\text{bool}) \in PC_2. \]

**Remark 5.6.** The last condition can also be expressed as follows: the diagram

\[
\begin{array}{ccc}
PC_1 & \xrightarrow{\tau^p_1} & S_1^t \times \{\text{bool}\} \\
\downarrow \sigma_4 & & \downarrow \sigma_1 \times \sigma_1 \\
PC_2 & \xrightarrow{\tau^p_2} & S_2^t \times \sigma_1(\{\text{bool}\})
\end{array}
\]

commutes.

**Definition 5.11.** *Aspect-Class signature morphism.*

Given an Aspect signature \( \Sigma_1 = \langle S_1, \leq_1, AC_1, ST_1, PC_1, \tau^A_1, \tau^S_1, \tau^p_1 \rangle \) and a class signature \( \Sigma_2 = \langle S_2, \leq_2, AC_2, ST_2, \tau^A_2, \tau^S_2 \rangle \), an *Aspect-Class signature morphism* \( \sigma: \Sigma_1 \rightarrow \Sigma_2 \) consists of a 4-tuple of maps \( (\sigma_1, \sigma_2, \sigma_3, \sigma_4) \) where \( \sigma_1: S_1 \rightarrow S_2, \sigma_2: AC_1 \rightarrow AC_2, \sigma_3: ST_1 \rightarrow ST_2, \) and \( \sigma_4: PC_1 \rightarrow AC_2 \) such that:

- \( (\sigma_1, \sigma_2, \sigma_3): \langle S_1, \leq_1, AC_1, ST_1, \tau^A_1, \tau^S_1 \rangle \rightarrow \langle S_2, \leq_2, AC_2, ST_2, \tau^A_2, \tau^S_2 \rangle \) is a class signature morphism.

- \( \forall pc: s_1 s_2 \ldots s_n \rightarrow \text{bool} \in PC_1, \; \sigma_4(pc): \sigma_1(s_1) \sigma_1(s_2) \ldots \sigma_1(s_n) \rightarrow \sigma_1(\text{bool}) \in AC_2. \)

**Remark 5.7.** The last condition can also be expressed as follows: the diagram

\[
\begin{array}{ccc}
PC_1 & \xrightarrow{\tau^p_1} & S_1^t \times \{\text{bool}\} \\
\downarrow \sigma_4 & & \downarrow \sigma_1 \times \sigma_1 \\
AC_2 & \xrightarrow{\tau^p_2} & S_2^t \times \sigma_1(\{\text{bool}\})
\end{array}
\]

commutes.

### 5.2.3.2 Specification Morphism

Let \( \sigma \) be a signature morphism and \( \varphi \) a formula of \( \mathcal{F} \). The translation or image of \( \varphi \) by \( \sigma \) is calculated by replacing in \( \varphi \) all the terms and actions with their respective images by \( \sigma \). If \( x \) is a variable, then its image by \( \sigma \) is \( x \). More, the logical symbols, the reserved words and the wildcards (in pointcut formulae) in \( \varphi \) must be unchanged in \( \sigma(\varphi) \). This translation strategy is the same as in [31] and [110].
**Definition 5.12.** *Image of a formula by a signature morphism.*

Let $\sigma : \Sigma_1 \to \Sigma_2$ be a signature morphism (it could be any kind of morphism) and $X$ a $S$-sorted set of variables. The image of a formula by $\sigma$ is computed as follows:

- $\forall x \in X, \sigma(x) = x$

- if $t_1$ is a term of sort $s_1, \ldots, t_n$ a term of sort $s_n$ and $f : s_1 \ldots s_n \to s$ is an action signature, then $\sigma(f(t_1, t_2, \ldots, t_n)) = \sigma(f)(\sigma(t_1), \sigma(t_2), \ldots, \sigma(t_n))$

- if $t_1$ is a term of sort $s_1, \ldots, t_n$ a term of sort $s_n$ and $f : s_1 \ldots s_n \to s$ is an attribute signature, then $\sigma(f(t_1, t_2, \ldots, t_n)) = \sigma(f)(\sigma(t_1), \sigma(t_2), \ldots, \sigma(t_n))$

- $\sigma(t_1 = t_2) = (\sigma(t_1) = \sigma(t_2))$

- $\sigma(t \to t_1 = t_2) = (\sigma(t) \to \sigma(t_1) = \sigma(t_2))$

- $\sigma(Gt) = G\sigma(t)$

- if $t_1$ is a term of sort $s_1, \ldots, t_n$ a term of sort $s_n$ and $f : s_1 \ldots s_n \to s$ is an action signature, $\sigma([f(t_1, t_2, \ldots, t_n)]t) = [\sigma(f(t_1, t_2, \ldots, t_n))]\sigma(t)$

- if $t_1$ is a term of sort $s_1, \ldots, t_n$ a term of sort $s_n$ and $f : s_1 \ldots s_n \to s$ is an action signature, $\sigma(Of(t_1, t_2, \ldots, t_n)) = O\sigma(f(t_1, t_2, \ldots, t_n))$

- if $t_1$ is a term of sort $s_1, \ldots, t_n$ a term of sort $s_n$ and $f : s_1 \ldots s_n \to s$ is an action signature, $\sigma(Pf(t_1, t_2, \ldots, t_n)) = P\sigma(f(t_1, t_2, \ldots, t_n))$

- if $t_1$ is a term of sort $s_1, \ldots, t_n$ a term of sort $s_n$ and $f : s_1 \ldots s_n \to s$ is an action signature, $\sigma(If(t_1, t_2, \ldots, t_n)) = I\sigma(f(t_1, t_2, \ldots, t_n))$

- similarly for the other modalities of $L_A$.

**Definition 5.13.** *ADT specification morphism.*

Let $\text{SPEC}_{\text{ADT}}_1 = (\Sigma_1, \text{VAR}_1, \text{AX}_1)$ and $\text{SPEC}_{\text{ADT}}_2 = (\Sigma_2, \text{VAR}_2, \text{AX}_2)$ be two ADT specifications. An ADT specification morphism $\sigma : \text{SPEC}_{\text{ADT}}_1 \to \text{SPEC}_{\text{ADT}}_2$ is an ADT signature morphism such that:

- $\forall \varphi \in \text{AX}_1, \sigma(\varphi)$ is a theorem in $\text{SPEC}_{\text{ADT}}_2$. 
**Definition 5.14. Class specification morphism.**
Let \( \text{SPEC\_CLASS}_1 = (\Sigma_1, \text{VAR}_1, \text{AX}_1, \text{PrAX}_1) \) and \( \text{SPEC\_CLASS}_2 = (\Sigma_2, \text{VAR}_2, \text{AX}_2, \text{PrAX}_2) \) be two Class specifications. An *Class specification morphism* \( \sigma: \text{SPEC\_CLASS}_1 \rightarrow \text{SPEC\_CLASS}_2 \) is a Class signature morphism such that:

- \( \forall \varphi \in \text{AX}_1, \sigma(\varphi) \) is a theorem in \( \text{SPEC\_CLASS}_2 \).
- \( \forall \varphi \in \text{PrAX}_1, \sigma(\varphi) \) is a prescription theorem in \( \text{SPEC\_CLASS}_2 \).

**Definition 5.15. Aspect specification morphism.**
Let \( \text{SPEC\_ASPECT}_1 = (\Sigma_1, \text{VAR}_1, \text{AX}_1, \text{PAX}_1, \text{AAX}_1, \text{PrAX}_1) \) and \( \text{SPEC\_ASPECT}_2 = (\Sigma_2, \text{VAR}_2, \text{AX}_2, \text{PAX}_2, \text{AAX}_2, \text{PrAX}_2) \) be two Aspect specifications.

An *Aspect specification morphism* \( \sigma: \text{SPEC\_ASPECT}_1 \rightarrow \text{SPEC\_ASPECT}_2 \) is an Aspect signature morphism such that:

- \( \forall \varphi \in \text{AX}_1, \sigma(\varphi) \) is a theorem in \( \text{SPEC\_ASPECT}_2 \).
- \( \forall \varphi \in \text{PAX}_1, \sigma(\varphi) \) is a pointcut theorem in \( \text{SPEC\_ASPECT}_2 \).
- \( \forall \varphi \in \text{AAX}_1, \sigma(\varphi) \) is an advice theorem in \( \text{SPEC\_ASPECT}_2 \).
- \( \forall \varphi \in \text{PrAX}_1, \sigma(\varphi) \) is a prescription theorem in \( \text{SPEC\_ASPECT}_2 \).

**Definition 5.16. Aspect-Class specification morphism.**
Let \( \text{SPEC\_ASPECT}_1 = (\Sigma_1, \text{VAR}_1, \text{AX}_1, \text{PAX}_1, \text{AAX}_1, \text{PrAX}_1) \) and \( \text{SPEC\_CLASS}_2 = (\Sigma_2, \text{VAR}_2, \text{AX}_2, \text{PrAX}_2) \) be respectively an Aspect and a Class specifications.

An *Aspect-Class specification morphism* \( \sigma: \text{SPEC\_ASPECT}_1 \rightarrow \text{SPEC\_CLASS}_2 \) is an Aspect-Class signature morphism such that:

- \( \forall \varphi \in \text{AX}_1, \sigma(\varphi) \) is a theorem in \( \text{SPEC\_CLASS}_2 \).
- \( \forall \varphi \in \text{PAX}_1, \sigma(\varphi) \) is an axiom in \( \text{SPEC\_CLASS}_2 \).
- \( \forall \varphi \in \text{AAX}_1, \sigma(\varphi) \) is an axiom in \( \text{SPEC\_CLASS}_2 \).
- \( \forall \varphi \in \text{PrAX}_1, \sigma(\varphi) \) is a prescription axiom or theorem in \( \text{SPEC\_CLASS}_2 \).

**Remark 5.8.** In practice, an Aspect-Class specification morphism will merely add each pointcut axiom, each advice axiom, and some prescription axioms of the aspect specification into the class specification.
5.3 Module specification

5.3.1 Module specification

Modules can be seen as the basic building blocks being used for modularization, which is one of the main principles in software development. We don’t fundamentally modify the definition of Ehrig and Mahr given in the section 2.4.3.

Definition 5.17. ADT Module Specification.

An ADT module specification is a module specification in which $PAR$, $IMP$, $EXP$, $BOD$ are ADT specifications and morphisms are ADT specification morphisms.

Definition 5.18. Class Module Specification.

A Class module specification is a module specification in which $PAR$, $IMP$, $EXP$, $BOD$ are Class specifications and morphisms are Class specification morphisms.


An aspect module specification is a module specification in which $PAR$, $IMP$, $EXP$, $BOD$ are Aspect specifications and morphisms are Aspect specification morphisms.

5.3.2 Module morphism

In section 2.4.3, we presented the definition 2.23 of module morphism $m : MOD_1 \rightarrow MOD_2$ between two modules $MOD_1$ and $MOD_2$. If $MOD_1$ and $MOD_2$ are ADT (resp. Class, Aspect) module specifications, then $m_p$, $m_e$, $m_i$, $m_b$ are ADT (resp. Class, Aspect) specification morphisms and $m$ is a ADT (resp. Class, Aspect) module morphism. If $MOD_1$ is an aspect module specification and $MOD_2$ is a class module specification, then $m_p$, $m_e$, $m_i$, $m_b$ are aspect-class specification morphisms and $m$ is an Aspect-Class module morphism.

5.4 Category of Module Specifications

One of our main motivations to use category theory is that it has means that allow composing two or more modules to get as result a larger module. In "Categorical Manifesto" [38], Goguen asserts several "dogmas", each of which suggests a use for a particular categorical construction. The dogma for the section on "Colimits" says:
Given a species of structure, say widgets, then the result of interconnecting a system of widgets to form a super-widget corresponds to taking the colimit of the diagram of widgets in which the morphisms show how they are interconnected.

Roughly speaking, what this means is that category theory prescribes a way in which one can depict each widget as a dot, and the relations that describe how they interconnect as arrows. The resulting diagram has a precise interpretation, and by applying the categorical operation called "colimit" to it, one can derive a description of the "super-widget" thus formed. Many computer scientists have used this idea to give a precise semantics to specification languages: to describe exactly how combining specifications of parts of a system results in a specification of the entire system [67]. To be able to perform such operations on a category, we have to show that this category is finitely co-complete. In the following, we note:

\( \mathbf{ADT}_{\text{Sig}} \) the set of all ADT signatures and the ADT signature morphisms of a system.

\( \mathbf{CLASS}_{\text{Sig}} \) the set of all Class signatures and the Class signature morphisms of a system.

\( \mathbf{ASPECT}_{\text{Sig}} \) the set of all Aspect signatures and the Aspect signature morphisms of a system.

\( \mathbf{ADT}_{\text{Spec}} \) the set of all ADT specifications and the ADT specification morphisms of a system.

\( \mathbf{CLASS}_{\text{Spec}} \) the set of all Class specifications and the Class specification morphisms of a system.

\( \mathbf{ASPECT}_{\text{Spec}} \) the set of all Aspect specifications and the Aspect specification morphisms of a system.

\( \mathbf{CAT}_{\text{Spec}} \) the set of all ADT, Class and Aspect specifications and their different morphisms of a system.

\( \mathbf{ADT}_{\text{Mod}} \) the set of all ADT module specifications and the ADT module morphisms of a system.

\( \mathbf{CLASS}_{\text{Mod}} \) the set of all Class module specifications and the Class module morphisms of a system.
\text{ASPECT}_{\text{Mod}}$ the set of all Aspect module specifications and the Aspect module morphisms of a system.

\text{CAT}_{\text{Mod}}$ the set of all ADT, Class and Aspect module specifications and their different morphisms of a system.

### 5.4.1 Category of Specifications

**Remark 5.9.** It is worth to notice that a Class specification is simply an Aspect specification with the Pointcut, Pointcut Axioms and Advice axioms fields empty. Also, an ADT specification is a Class specification with the States, Initial, and Prescription Axioms fields empty.

**Theorem 5.1.** $\text{ADT}_{\text{Sig}}$, $\text{CLASS}_{\text{Sig}}$, and $\text{ASPECT}_{\text{Sig}}$ form categories.

**Proof.** :

It is easy to show that $\text{ADT}_{\text{Sig}}$, $\text{CLASS}_{\text{Sig}}$, and $\text{ASPECT}_{\text{Sig}}$ satisfy the constraints of the definition 2.3.

**Theorem 5.2.** $\text{ADT}_{\text{Spec}}$, $\text{CLASS}_{\text{Spec}}$, and $\text{ASPECT}_{\text{Spec}}$ form categories.

**Proof.** :

It is easy to show that $\text{ADT}_{\text{Spec}}$, $\text{CLASS}_{\text{Spec}}$, and $\text{ASPECT}_{\text{Spec}}$ satisfy the constraints of the definition 2.3.

**Theorem 5.3.** $\text{ADT}_{\text{Sig}}$ forms a finitely co-complete category.

**Proof.** Recall that $AC = \bigcup_{s \in S^*, \omega \in S} (AC_{\omega,s})$ is a set of action signatures where $\{AC_{\omega,s}\}_{\omega \in S^*, s \in S}$ is a finite $S^* \times S$-sorted set of pairwise disjoint subsets of $AC$, and $\tau$ is an action profile function s.t. $\tau : AC \rightarrow S^* \times S$. Let $AC$ be the set of all possible $AC$. Let $\text{Set}$ be the category of sets, $\text{FSet}$ be the category of finite sets, and $\text{FPoSet}$ the category of finite posets. It is easy to show that $AC$ is a full subcategory of $\text{FSet}$ (by using the definition 2.8). We need to show that $AC$ is a reflexive subcategory of $\text{FSet}$. Let $S$ be an object of $\text{FSet}$. Let $AC = \bigcup_{s \in S} (AC_{\lambda,s})$ where $\lambda$ is the empty string of $S^*$. $AC$ is then an object of $AC$ and $k : S \rightarrow AC$ s.t. $k(s) = \sigma : \lambda \rightarrow s$ is a morphism of $\text{FSet}$. Let $AC' = \bigcup_{s \in S} (AC'_{\omega,s})$ be an object of $AC$ and $f : S \rightarrow AC'$ a morphism of $\text{FSet}$. Let $f' : AC \rightarrow AC'$ s.t. $f' (\sigma : \lambda \rightarrow s) = \sigma : \lambda \rightarrow s$. We can easily
verify that \( f = f' \circ k \) and \( f' \) is the unique morphism s.t. \( f = f' \circ k \). This then shows that \( AC \) is reflexive. Now, let \( L : AC \rightarrow FSet \) be the inclusion functor. \( L \) is finitely co-continuous (sums of finite sets and co-equalizers of finite sets are finite sets). Let \( R : FPoset \rightarrow FSet \) be the functor defined by \( R((S, \leq)) = S^* \times S, R(f) = f^* \times f \).

The comma category \( (L \downarrow R) \) has as objects structures of the form \( \langle AC, \langle S, \leq \rangle, \tau \rangle \) where \( \tau : AC \rightarrow S^* \times S \). A morphism from \( \langle AC, \langle S, \leq \rangle, \tau \rangle \) to \( \langle AC', \langle S', \leq' \rangle, \tau' \rangle \) is an ADT signature morphism. That means that the category \( \mathbf{ADTSig} \) is isomorphic to \( (L \downarrow R) \). \( FSet \) is finitely cocomplete and \( AC \) is a full reflexive subcategory of \( FSet \); thus \( AC \) is finitely cocomplete (see theorem 2.2). \( AC \) and \( FPoset \) are finitely cocomplete and \( L \) is finitely co-continuous. That implies by proposition 2.2 that \( (L \downarrow R) \) is finitely cocomplete. Whence, \( \mathbf{ADTSig} \) is finitely cocomplete. \( \square \)

**Theorem 5.4.** \( \mathbf{CLASS}_{\mathbf{Sig}} \) forms a finitely cocomplete category.

**Proof.** Recall that \( \mathbf{ST} = \bigcup_{\omega \in S^*, s \in S} \langle \mathbf{ST}_{\omega,s} \rangle \) is a set of attribute signatures where \( \{\mathbf{ST}_{\omega,s}\}_{\omega \in S^*, s \in S} \) is a \( S^* \times S \)-sorted set of pairwise disjoint subsets of \( \mathbf{ST} \), and \( \tau^S \) is an attribute profile function s.t. \( \tau^S : \mathbf{ST} \rightarrow S^* \times S \). Let \( \mathbf{ST} \) be the set of all possible \( \mathbf{ST} \). It is easy to show that \( \mathbf{ST} \) is a full subcategory of \( FSet \) (by using the definition 2.8). By the same the reasoning in the precedent proof, \( \mathbf{ST} \) is a reflexive subcategory of \( FSet \). And thus, \( \mathbf{ST} \) is finitely cocomplete by the theorem 2.2. Let \( F : \mathbf{ST} \rightarrow \mathbf{Set} \) be the inclusion functor. \( F \) is finitely co-continuous. Let us now define the functor \( \Delta' : \mathbf{ADTSig} \rightarrow \mathbf{Set} \) as follows.

\[
\Delta'\langle AC, S, \leq, \tau^A \rangle = S^* \times S
\]

\[
\Delta'\langle \sigma_2, \sigma_1 \rangle = \sigma_1^* \times \sigma_1
\]

The objects of \( (F \downarrow \Delta') \) are triples \( \langle \mathbf{ST}, \Sigma, \tau^S \rangle \) such that \( \tau^S : \mathbf{ST} \rightarrow \Delta'(\Sigma) \) with \( \Sigma = \langle AC, S, \leq, \tau^A \rangle \). The morphisms of \( (F \downarrow \Delta') \) from \( \langle \mathbf{ST}, \Sigma, \tau^S \rangle \) to \( \langle \mathbf{ST}', \Sigma', \tau'^S \rangle \) are pairs \( (u, v) \) such that \( u : \mathbf{ST} \rightarrow \mathbf{ST}' \) and \( v : \Sigma \rightarrow \Sigma' \). If we write \( \sigma_3 \) for \( u, \langle \sigma_2, \sigma_1 \rangle \) for \( v \) where \( \Sigma = \langle AC, S, \leq, \tau^A \rangle \) and \( \Sigma' = \langle AC', S', \leq', \tau'^A \rangle \), then the condition for having a morphism in the comma category writes as

\[
\begin{array}{ccc}
\mathbf{ST} & \xrightarrow{\sigma_3} & \mathbf{ST}' \\
\tau^S \downarrow \quad & & \downarrow \tau'^S \\
S^* \times S & \xrightarrow{\sigma_1^* \times \sigma_1} & S'^* \times S'
\end{array}
\]
which is nothing but the main condition of a morphism of class signatures (i.e., the one that distinguishes a morphism of ADT signatures from a morphism of class signatures). That gives the isomorphism of categories

$$\text{CLASS}_{\text{Sig}} \simeq (F \downarrow \Delta').$$

Knowing that $\text{ADT}_{\text{Sig}}$ is finitely cocomplete (by the previous subsection), $F$ is finitely co-continuous, and $ST$ is finitely cocomplete, that implies that $(F \downarrow \Delta')$ is finitely cocomplete by proposition 2.2; that implies that $\text{CLASS}_{\text{Sig}}$ is finitely co-complete. □

**Theorem 5.5.** $\text{ASPECT}_{\text{Sig}}$ forms a finitely cocomplete category.

**Proof.** Recall that $PC = \bigcup_{\omega \in S^*} \{PC_\omega\}$ is a finite set of pointcut signatures with bool the boolean sort and where $\{PC_\omega\}_{\omega \in S^*}$ is a $S^*$-sorted set of pairwise disjoint subsets of $PC$, and $\tau^p$ is a pointcut profile function s.t $\tau^p : PC \to S^* \times \text{bool}$ and if $np : s_1, s_2, \ldots, s_n \to \text{bool} \in PC$, $\tau^p(np) = \langle s_1, s_2, \ldots, s_n, \text{bool} \rangle$. Let $PC$ be the set of all possible $PC$. It is easy to show that $PC$ is a full subcategory of $FSet$ (by using the definition 2.8). By the same the reasoning in the precedent proofs, $PC$ is a reflexive subcategory of $FSet$. And thus, $PC$ is finitely cocomplete by the theorem 2.2, since $FSet$ is finitely cocomplete. Let $F' : PC \to \text{Set}$ be the inclusion functor. $F'$ is finitely co-continuous. Let us now define the functor $\Delta'' : \text{CLASS}_{\text{Sig}} \to \text{Set}$ as follows.

$$\Delta''(\langle AC, ST, S, \leq, \tau^1, \tau^5 \rangle) = S^* \times \{\text{bool}\}$$

$$\Delta''(\langle \sigma_1, \sigma_2, \sigma_3 \rangle) = \sigma^*_1 \times \sigma_1$$

The objects of $(F' \downarrow \Delta'')$ are triples $\langle PC, \Sigma, \tau^p \rangle$ such that $\tau^p : PC \to \Delta''(\Sigma)$ with $\Sigma = \langle AC, ST, S, \leq, \tau^1, \tau^5 \rangle$. The morphisms of $(F' \downarrow \Delta'')$ from $\langle PC, \Sigma, \tau^p \rangle$ to $\langle PC', \Sigma', \tau'^p \rangle$ are pairs $(u, v)$ such that $u : PC \to PC'$ and $v : \Sigma \to \Sigma'$. If we write $\sigma_4$ for $u$, $\langle \sigma_1, \sigma_2, \sigma_3 \rangle$ for $v$ where $\Sigma = \langle AC, ST, S, \leq, \tau^1, \tau^5 \rangle$ and $\Sigma' = \langle AC', ST', S', \leq', \tau'^1, \tau'^5 \rangle$, then the condition for having a morphism in the comma category writes as

$$\begin{array}{ccc}
PC & \xrightarrow{\sigma_4} & PC' \\
\tau^p \downarrow & & \downarrow \tau'^p \\
S^* \times \{\text{bool}\} & \xrightarrow{\sigma_1^* \times \sigma_1} & S'^* \times \{\text{bool}\}
\end{array}$$
which is nothing but the main condition that characterizes a morphism of aspect signatures (i.e., the one that distinguishes a morphism of class signatures from a morphism of aspect signatures). That gives the isomorphism of categories

\[ \text{ASPECT}_{\text{Sig}} \simeq (F' \downarrow \Delta''). \]

Knowing that \( \text{CLASS}_{\text{Sig}} \) is finitely co-complete (by the previous subsection), \( F' \) is finitely co-continuous, and \( PC \) is finitely co-complete, that implies that \( (F' \downarrow \Delta'') \) is finitely co-complete by proposition 2.2; that implies that \( \text{ASPECT}_{\text{Sig}} \) is finitely co-complete. \( \square \)

**Theorem 5.6.** \( \text{ADT}_{\text{Spec}} \) forms a finitely co-complete category.

**Theorem 5.7.** \( \text{CLASS}_{\text{Spec}} \) forms a finitely co-complete category.

**Theorem 5.8.** \( \text{ASPECT}_{\text{Spec}} \) forms a finitely co-complete category.

The proofs of the theorems 5.6, 5.7, and 5.8 are the same as one in [110] (chapter 4, pp 89-90) or [40] (pp 27-28) (since a specification is obtained by adding the variable and axiom components).

### 5.4.2 Category of Module Specifications

**Theorem 5.9.** \( \text{ADT}_{\text{Mod}}, \text{CLASS}_{\text{Mod}}, \) and \( \text{ASPECT}_{\text{Mod}} \) form categories.

**Proof.** It is easy to show that \( \text{ADT}_{\text{Mod}}, \text{CLASS}_{\text{Mod}}, \) and \( \text{ASPECT}_{\text{Mod}} \) are categories since a module specification is composed of four algebraic specifications. \( \square \)

**Theorem 5.10.** \( \text{ADT}_{\text{Mod}}, \text{CLASS}_{\text{Mod}}, \) and \( \text{ASPECT}_{\text{Mod}} \) are finitely co-complete.

**Proof.** We will prove that \( \text{ADT}_{\text{Mod}} \) is finitely co-complete. The proof for \( \text{CLASS}_{\text{Mod}}, \) and \( \text{ASPECT}_{\text{Mod}} \) is obtained by the same reasoning. We will use the theorem 2.1 to prove this theorem. We need to show that \( \text{ADT}_{\text{Mod}} \) has an initial object, coproducts and coequalizers.

#### 5.4.2.0.1 Existence of initial object

Let \( M = (P, E, I, B, e, s, i, v) \) be an ADT module specification. That implies that \( P, E, I, B \) are objects of \( \text{ADT}_{\text{Spec}} \) which has an initial object \( X \), since \( \text{ADT}_{\text{Mod}} \) is cocomplete. Thus, there exist \( g_1, g_2, g_3, g_4 \) s.t. \( g_1 : X \rightarrow P, g_2 : X \rightarrow E, g_3 : X \rightarrow I, \)
\( g_4 : X \rightarrow B \) and \( g_i, i = 1, 2, 3, 4 \) are unique. Let \( M_0 = (X, X, X, e_0, s_0, i_0, v_0) \) and 
\[ g = (g_1, g_2, g_3, g_4). \]
\( g \) is an unique morphism from \( M_0 \) to \( M \). \( e_0, s_0, i_0, v_0 \) are such that the following diagram commutes:

![Diagram](attachment:image.png)

Hence \( M_0 \) is the initial object of \( \text{ADT}_{\text{Spec}} \).

5.4.2.0.2 Existence of coproducts

Let \( M_1 = (P_1, E_1, I_1, B_1, e_1, s_1, i_1, v_1) \) and \( M_2 = (P_2, E_2, I_2, B_2, e_2, s_2, i_2, v_2) \) be two ADT module specifications. Let \( M_3 = (P_3, E_3, I_3, B_3, e_3, s_3, i_3, v_3) \) s.t.

- \( P_3 = P_1 + P_2 \) with \( i^p_1 : P_1 \rightarrow P_3 \) and \( i^p_2 : P_2 \rightarrow P_3 \)
- \( E_3 = E_1 + E_2 \) with \( i^e_1 : E_1 \rightarrow E_3 \) and \( i^e_2 : E_2 \rightarrow E_3 \)
- \( I_3 = I_1 + I_2 \) with \( i^i_1 : I_1 \rightarrow I_3 \) and \( i^i_2 : I_2 \rightarrow I_3 \)
- \( B_3 = B_1 + B_2 \) with \( i^b_1 : B_1 \rightarrow B_3 \) and \( i^b_2 : B_2 \rightarrow B_3 \)
$e_3, s_3, i_3, v_3$ are such that the following diagram commutes ($i_{M_1} = \langle i^p_1, i^E_1, i^I_1, i^B_1 \rangle$ and $i_{M_2} = \langle i^p_2, i^E_2, i^I_2, i^B_2 \rangle$ are morphisms from $M_1$ to $M_3$ and $M_2$ to $M_3$):

Let $M = (P, E, I, B)$ be an arbitrary ADT module specification, and $f : M \to M_1$ and $g : M \to M_2$ s.t. $f = \langle f_P, f_E, f_I, f_B \rangle$ and $g = \langle g_P, g_E, g_I, g_B \rangle$ and $f_P : P_1 \to P$, $f_E : E_1 \to E$, $f_I : I_1 \to I$, $f_B : B_1 \to B$ and $g_P : P_2 \to P$, $g_E : E_2 \to E$, $g_I : I_2 \to I$, $g_B : B_2 \to B$. We know that exist $h_P, h_E, h_I, h_B$ which are unique and such that the following diagrams commute:

Let $h = \langle h_P, h_E, h_I, h_B \rangle$. $h$ is the unique morphism from $M_3$ to $M$. That proves that $M_3$ and $i_{M_1} = \langle i^p_1, i^E_1, i^I_1, i^B_1 \rangle$ and $i_{M_2} = \langle i^p_2, i^E_2, i^I_2, i^B_2 \rangle$ constitute the coproduct of $M_1$ and $M_2$. 
5.4.2.0.3 Existence of coequalizers

Let $M_1 = (P_1, E_1, I_1, B_1, e_1, s_1, i_1, v_1)$ and $M_2 = (P_2, E_2, I_2, B_2, e_2, s_2, i_2, v_2)$ be two ADT module specifications. Let $f, g : M_1 \to M_2$ two parallel ADT module morphisms s.t. $f = \langle f_P, f_E, f_I, f_B \rangle$ and $g = \langle g_P, g_E, g_I, g_B \rangle$. Let $Z = (Z_P, Z_E, Z_I, Z_B, e_Z, s_Z, i_Z, v_Z)$ and $e = \langle e_P, e_E, e_I, e_B \rangle$ s.t. (since $\mathsf{ADT}_{\text{Spec}}$ is finitely co-complete.)

$Z_P$ and $e_P : P_2 \to Z_P$ constitute the coequalizer of $f_P, g_P : P_1 \to P_2$ (1)

$Z_E$ and $e_E : E_2 \to Z_E$ constitute the coequalizer of $f_E, g_E : E_1 \to E_2$ (2)

$Z_I$ and $e_I : I_2 \to Z_I$ constitute the coequalizer of $f_I, g_I : I_1 \to I_2$ (3)

$Z_B$ and $e_B : B_2 \to Z_B$ constitute the coequalizer of $f_B, g_B : B_1 \to B_2$ (4)

$e_Z, s_Z, i_Z, v_Z$ are such that the diagram $M_2 \xrightarrow{e} Z$ commutes. We want to show that $Z$ and $e : M_2 \to Z$ constitute the coequalizer of $f$ and $g$.

$$e \circ f = \langle e_P \circ f_P, e_E \circ f_E, e_I \circ f_I, e_B \circ f_B \rangle$$

$$= \langle e_P \circ g_P, e_E \circ g_E, e_I \circ g_I, e_B \circ g_B \rangle \text{ because of (1), (2), (3), (4)}$$

$$= e \circ g$$

Let $V = (V_P, V_E, V_I, V_B)$ be an ADT module specification and $e' = \langle e'_P, e'_E, e'_I, e'_B \rangle : M_2 \to V$ s.t. $e' \circ f = e' \circ g$. We need to show that there exists an unique morphism $k : Z \to V$ of $\mathsf{ADT}_{\text{Mod}}$ s.t. $k \circ e = e'$, i.e., the following diagram commutes:

$$\begin{array}{ccc}
M_1 & \xrightarrow{f} & M_2 \\
\downarrow g & & \downarrow e \\
 & \searrow k & \\
 & & V
\end{array}$$

$e'_P : P_2 \to V_P$ implies that there exists an unique morphism $k_P : Z_P \to V_P$ of $\mathsf{ADT}_{\text{Spec}}$ s.t. $k_P \circ e_P = e'_P$ because of (1).

$e'_E : E_2 \to V_E$ implies that there exits an unique morphism $k_E : Z_E \to V_E$ of $\mathsf{ADT}_{\text{Spec}}$ s.t. $k_E \circ e_E = e'_E$ because of (2).

$e'_I : I_2 \to V_I$ implies that there exits an unique morphism $k_I : Z_I \to V_I$ of $\mathsf{ADT}_{\text{Spec}}$ s.t. $k_I \circ e_I = e'_I$ because of (3).
\( e'_B : B_2 \to V_B \) implies that there exits an unique morphism \( k_B : Z_B \to V_B \) of \( \text{ADT}^{\text{Spec}} \) s.t. \( k_B \circ e_B = e'_B \) because of (4).

Let \( k = \langle k_P, k_E, k_I, k_B \rangle \). Thus, \( k : Z \to V, k \circ e = e' \), and \( k \) is unique. Whence, \( Z \) and \( e : M_2 \to Z \) constitute the coequalizer of \( f \) and \( g \).

\( \text{ADT}^{\text{Mod}} \) has an initial object, coproducts and coequalizers; thus \( \text{ADT}^{\text{Mod}} \) is finitely cocomplete (see theorem 2.1). Hence, \( \text{CLASS}^{\text{Mod}} \) and \( \text{ASPECT}^{\text{Mod}} \) are also finitely cocompletes. A consequence of this finite co-completeness is that we can calculate the colimit of any finite diagram of the system. \( \square \)

**Remark 5.10.** Because of the remark 5.9, we can assume theoretically that \( \text{ADT}^{\text{Spec}} \) and \( \text{CLASS}^{\text{Spec}} \) are subcategories of \( \text{ASPECT}^{\text{Spec}} \); even if, conceptually it is not the case. Thus, we can assume (without losing generality) that \( \text{CAT}^{\text{Spec}} \cong \text{ASPECT}^{\text{Spec}} \) and infer that \( \text{CAT}^{\text{Spec}} \) is finitely cocomplete. Similarly, we can assume that \( \text{CAT}^{\text{MOD}} \cong \text{ASPECT}^{\text{MOD}} \) and infer that \( \text{CAT}^{\text{MOD}} \) is finitely cocomplete.

### 5.5 Semantics

In the last two decades, universal algebra has become useful and important in theoretical computer science. The applications of universal algebra in theoretical computer science have increased more and more. Most of them deal with formal semantics. By splitting the syntactic and semantic part of a data type, a program or any other object of interest, universal algebra provides methods to define and argue about semantics in a suitable formal framework [107].

In this approach, syntactic and semantics domains are uncoupled. ADT (resp. Class, Aspect) algebraic specifications are syntactically defined and are interpreted in the semantics domain by using mathematical algebras or models.

An algebraic specification is defined as a signature and some axioms or formulae. For a given signature \( \Sigma \), a \( \Sigma \)-algebra is defined. A model of a given algebraic specification is a \( \Sigma \)-algebra validating the axioms. Thus, each algebraic specification induces a class of models. We use the concept algebras and the logic \( L_A \) (defined in chapter 4) to define our semantics.
5.5.1 Semantics of Specifications

An algebra is a set together with a family of operations. In this section, we define the semantics of ADT, Class and Aspect specifications. We chose to use a loose semantics because it is quite natural and simple. Loose means that an algebraic specification is denoted by a set of models which is determined by the given abstract properties. Our approach is based on a denotational semantics because it gives a precise mathematical meaning to a program in a way that is as conceptually clear and simple as possible, and that supports proving properties of programs.

5.5.1.1 Σ-Algebra

The basic idea is that a sort denotes a set of data items of that sort, and an operation denotes a function from input data items (of the appropriate sorts) to output data items.

**Example 5.8.** The assignment operation \( a : v e \rightarrow p \) takes a variable of sort \( v \) (i.e., the variables) and an expression of sort \( e \) (i.e., the expressions) as its inputs and produces as its output of sort \( p \). This assignment operation denotes a function \( f_a : A_v A_e \rightarrow A_p \) where \( A_v \) is the set of items of sort \( v \) in some algebra \( A \), \( A_e \) its set of items of sort \( e \) and \( A_p \) its set of items of sort \( p \).

**Definition 5.20.** ADT Σ-Algebra. Let \( \Sigma = \langle S, \leq, AC \rangle \) an ADT signature. An ADT Σ-Algebra \( \mathcal{A} \) is a pair \( (|\mathcal{A}|, m^A_{\mathcal{A}}) \) where:

- \(|\mathcal{A}| \) is a nonempty set called the carrier or domain of \( \mathcal{A} \), together with an indexed set \( \{|\mathcal{A}|_s, s \in S\} \) of subsets of \(|\mathcal{A}| \) s.t. \(|\mathcal{A}| = \bigcup_{s \in S} (|\mathcal{A}|_s) \) and if \( s \leq s' \) then \(|\mathcal{A}|_s \subseteq |\mathcal{A}|_{s'}

- \( m^A_{\mathcal{A}} \) is an action meaning function that assigns to each n-ary action signature \( f : s_1 s_2 \ldots s_n \rightarrow s \in AC \), a partial function \( m^A_{\mathcal{A}}(f) = f^\mathcal{A} : |\mathcal{A}|_{s_1} \times |\mathcal{A}|_{s_2} \times \ldots \times |\mathcal{A}|_{s_n} \rightarrow |\mathcal{A}|_s \) s.t. \( f: s_1 s_2 \ldots s_n \rightarrow s, f: s_1' s_2' \ldots s_n' \rightarrow s' \in AC \) and \( s_1 s_2 \ldots s_n s \leq s_1' s_2' \ldots s_n' s' \) implies \( f^\mathcal{A} : |\mathcal{A}|_{s_1} \times |\mathcal{A}|_{s_2} \times \ldots \times |\mathcal{A}|_{s_n} \rightarrow |\mathcal{A}|_s \) and \( f^\mathcal{A} : |\mathcal{A}|_{s_1} \times |\mathcal{A}|_{s_2} \times \ldots \times |\mathcal{A}|_{s_n} \rightarrow |\mathcal{A}|_{s'}. \)

In the sequel, \(|\mathcal{A}|_{s_1} \times |\mathcal{A}|_{s_2} \times \ldots \times |\mathcal{A}|_{s_n} \) is written \(|\mathcal{A}|_{\omega} \) where \( \omega = s_1 s_2 \ldots s_n \).
**Definition 5.21. Class Σ-Algebra.**

Let \( \Sigma = \langle S, \leq, AC, ST \rangle \) a Class signature. A Class Σ-Algebra \( \mathcal{A} \) is a triple \( \langle |\mathcal{A}|, m^A_{\mathcal{A}}, m^S_{\mathcal{A}} \rangle \) where:

- \( \langle |\mathcal{A}|, m^A_{\mathcal{A}} \rangle \) is an ADT Σ-Algebra.
- \( m^S_{\mathcal{A}} \) is an attribute meaning function that assigns to each \( n \)-ary attribute signature \( f: s_1 s_2 \ldots s_n \rightarrow s \in ST \), a partial function \( m^S_{\mathcal{A}}(f) = f^\mathcal{A} : |\mathcal{A}|_{s_1} \times |\mathcal{A}|_{s_2} \times \ldots \times |\mathcal{A}|_{s_n} \times W \rightarrow |\mathcal{A}|_s \) s.t. \( f: \omega \rightarrow s, f: \omega' \rightarrow s' \in ST \) and \( \omega, s \leq \omega', s' \) implies \( f^\mathcal{A} : |\mathcal{A}|_\omega \times W \rightarrow |\mathcal{A}|_s \) and \( f^\mathcal{A} : |\mathcal{A}|_\omega \times W \rightarrow |\mathcal{A}|_{s'} \) are equal on \( |\mathcal{A}|_\omega \times W \). \( W \) is a set of states (times).

We use \( W \) to define \( m^S_{\mathcal{A}} \) because this function is state dependent since the value of an attribute may change from one state to the other.

**Definition 5.22. Aspect Σ-Algebra.**

Let \( \Sigma = \langle S, \leq, AC, ST, PC \rangle \) an Aspect signature. An Aspect Σ-Algebra \( \mathcal{A} \) is a tuple \( \langle |\mathcal{A}|, m^A_{\mathcal{A}}, m^S_{\mathcal{A}}, m^P_{\mathcal{A}} \rangle \) where:

- \( \langle |\mathcal{A}|, m^A_{\mathcal{A}}, m^S_{\mathcal{A}} \rangle \) is a Class Σ-Algebra \( \mathcal{A} \).
- \( m^P_{\mathcal{A}} \) is a pointcut meaning function that assigns to each \( n \)-ary pointcut signature \( np: s_1 s_2 \ldots s_n \rightarrow \text{bool} \in PC \), a partial function \( m^P_{\mathcal{A}}(np) = np^\mathcal{A} : |\mathcal{A}|_{s_1} \times |\mathcal{A}|_{s_2} \times \ldots \times |\mathcal{A}|_{s_n} \rightarrow \text{BOOL} \) s.t. \( np: \omega \rightarrow \text{bool}, np: \omega' \rightarrow \text{bool} \in PC \) and \( \omega \leq \omega' \) implies \( np^\mathcal{A} : |\mathcal{A}|_\omega \rightarrow \text{BOOL} \) and \( np^\mathcal{A} : |\mathcal{A}|_{\omega'} \rightarrow \text{BOOL} \) are equal on \( |\mathcal{A}|_\omega \).

\( \text{BOOL} \) is the boolean algebra.

**Definition 5.23. Term Algebras**

For any signature \( \Sigma \) (ADT, Class, or Aspect signature), the set of terms \( T_\Sigma(X) \) forms a \( \Sigma \)-algebra, where each \( f \in \Sigma \) is given the syntactic interpretation:

\[
f^{T_\Sigma(X)} : \quad T_\Sigma(X)^n \quad \rightarrow \quad T_\Sigma(X) \quad (t_1, \ldots, t_n) \quad \mapsto \quad f(t_1, \ldots, t_n)
\]

The algebra \( T_\Sigma(X) \) is called an ADT (resp. Class, Aspect) term algebra if \( \Sigma \) is an ADT (resp. Class, Aspect) signature.

**Definition 5.24. Subalgebras**

Let \( \mathcal{A} \) and \( \mathcal{B} \) be two \( \Sigma \)-algebras. \( \mathcal{B} \) is a subalgebra of \( \mathcal{A} \) if \( |\mathcal{B}| \subseteq |\mathcal{A}| \) and \( f^\mathcal{B} = f^\mathcal{A} / |\mathcal{B}| \) for all \( n \)-ary \( f \in \Sigma \), where \( f^\mathcal{A} / |\mathcal{B}| \) is the restriction of \( f^\mathcal{A} \) on \( |\mathcal{B}|^n \).
Homomorphisms are structure-preserving functions between $\Sigma$-algebras.

**Definition 5.25.** $\Sigma$-Homomorphism

Let $\mathcal{A}$ and $\mathcal{B}$ be two $\Sigma$-algebras where $\Sigma$ is a signature with sort set $S$. A $\Sigma$-Homomorphism $h : \mathcal{A} \rightarrow \mathcal{B}$ is a $S$-sorted function $h = (h_s)_{s \in S} : |\mathcal{A}| \rightarrow |\mathcal{B}|$ such that for any $a_1 \in |\mathcal{A}|_{s_1}, \ldots, a_n \in |\mathcal{A}|_{s_n}$ and $n$-ary $f \in \Sigma$,

$$h(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$$

and

$$s \leq s' \text{ and } a \in A_s \text{ empty } h_s(a) = h_{s'}(a)$$

If $\mathcal{A}$ and $\mathcal{B}$ are ADT (resp. Class, Aspect) $\Sigma$-algebras, then $h$ is ADT (resp. Class, Aspect) $\Sigma$-Homomorphism. If $\mathcal{A}$ is an Aspect $\Sigma$-algebra and $\mathcal{B}$ is a Class $\Sigma$-algebra, then $h$ is an Aspect-Class $\Sigma$-Homomorphism.

**Definition 5.26.** Isomorphic Algebras.

Let $\mathcal{A}$ and $\mathcal{B}$ be two $\Sigma$-algebras. $\mathcal{A}$ and $\mathcal{B}$ are isomorphic, denoted by $\mathcal{A} \cong \mathcal{B}$ iff there exist homomorphisms $h : \mathcal{A} \rightarrow \mathcal{B}$ and $h' : \mathcal{B} \rightarrow \mathcal{A}$ such that $h \circ h'$ is the identity on the carrier set $|\mathcal{B}|$ and $h' \circ h$ is the identity on the carrier set $|\mathcal{A}|$.

**Definition 5.27.** Valuation

Let $\Sigma$ be a signature and $X$ a set of variables and $\mathcal{A}$ a $\Sigma$-algebra. A homomorphism $u : T_{\Sigma}(X) \rightarrow \mathcal{A}$ is called a valuation. The restriction $u/X$ of $u$ on $X$ is called assignment.

**Definition 5.28.** Substitution

A homomorphism $u : T_{\Sigma}(X) \rightarrow T_{\Sigma}(Y)$ defined from one term algebra to another is called a substitution.

The value of the substitution $u$ applied to a term $t \in T_{\Sigma}(X)$ is the term in $T_{\Sigma}(Y)$ obtained by substituting the term $u(x)$ for all occurrences of $x$ in $t$ simultaneously. For all $x \in X$:

$$u(t) = t[x/u(x)|x \in X]$$

### 5.5.1.2 Semantics of Class and Aspect Specifications

**5.5.1.2.1 Interpretation Structures** Let $\Sigma$ be a signature. $\Sigma$ will be interpreted on a structure $M = \langle W, \mathcal{U}, r, l, f, p \rangle$, where $W$ is a set of states (a state is defined by the values of all attributes at a given time); $\mathcal{U}$ is a $\Sigma$-algebra; if $\Sigma$ is an Class (resp. Aspect) signature, then $\mathcal{U}$ is a Class (resp. Aspect) $\Sigma$-algebra.
\[ r : \mathcal{A}_t \rightarrow 2^{W \times W} \]
\[ a \mapsto r(a) \]
is a function which associates to each atomic action \( a \), a binary relation \( r(a) \) s.t. \( \forall x, y \in W, x \ r(a) \ y \) iff there exist an execution of \( a \) from the state \( x \) to the state \( y \);

\[ l : \Phi_0 \rightarrow 2^W \]
\[ p \mapsto l(p) \]
is a function which associates to each atomic proposition \( p \) a subset \( l(p) \subseteq W \) s.t. \( \forall x \in W, x \in l(p) \) iff \( p \) holds in \( x \); functions \( r \) and \( l \) are inductively extended to the complex actions and formulae. The function \( fj : J \rightarrow AC \) associates to a joinpoint \( jp \), the action \( fj(jp) \) encapsulated in the body of \( jp \); the function \( pr : J \rightarrow \Phi_0 \) associates to a joinpoint \( jp \), the (atomic) proposition "the joinpoint \( jp \) is reached". The functions \( r, fj, pr \) and \( l \) are the same functions defined in the logic \( L_A \) (see chapter 4). \( \Phi_0 \) a set of atomic propositions denoted by \( p, q, \ldots \)

This model differs from more conventional ones, for example one in \([24, 25]\) where semantics is defined by an interface semantics (constituted of categories of parameter, import, and export with two functors), a construction semantics (defined by a free functor from the category of import models to the category of body models), and a behavior semantics represented by a functor between the categories of import and export models.

We think that this semantics is too complex. Our model uses the semantics model of the logic \( L_A \) and has advantages that allows us to formalize information like error recovery through corrective action or sanctions if desired. More, it integrates our prevention mechanism which allows to avoid undesirable behaviors in aspect-oriented systems.

In this structure, we can distinguish between description and prescription of behavior. Recall that action prescriptions are means to convey when actions may or must occur, through the deontic concepts of obligatory and permissible actions. The description is achieved by the traditional pre and post condition style description of actions. This then allows us to state when actions may and must happen as opposed to just describing the effects of such actions.

5.5.1.2.2 Term Interpretation Given a structure \( M = \langle W, \mathcal{U}, r, l, fj, pr \rangle \), the set \( T_E(X) \) of all terms over a signature \( \Sigma \) and a \( S \)-sorted set of variables \( X \), and a valuation \( u : T_E(X) \rightarrow \mathcal{U} \), the interpretation \( u(t)(w) \) of a term \( t \) in \( M \) (\( w \in W \)) is inductively computed as follows:
• For any constant $c$ of sort $s$ ($c : \lambda \to s$), $u(c)(w) = u(c) = c^\mathcal{U} \in \mathcal{U}_s$;

• For any variable $x$ of sort $s$, $u(x)(w) = u(x) \in \mathcal{U}_s$; note that a variable can be considered (in programming terms) as a nullary attribute symbol.

• If $t_1$ is a term of sort $s_1, \ldots, t_n$ a term of sort $s_n$ and $f : s_1\ldots s_n \to s$ is an action signature, then $u(f(t_1, t_2, \ldots, t_n))(w) = f^\mathcal{U}(u(t_1)(w), u(t_2)(w), \ldots, u(t_n)(w))$ if all $u(t_i)(w)$ are defined and undefined otherwise; $f^\mathcal{U}$ is the image of $f$ by the action meaning function $m^A_\mathcal{U}$;

• If $t_1$ is a term of sort $s_1, \ldots, t_n$ a term of sort $s_n$ and $f : s_1\ldots s_n \to s$ is an attribute signature, then $u(f(t_1, t_2, \ldots, t_n))(w) = f^\mathcal{U}(u(t_1)(w), u(t_2)(w), \ldots, u(t_n)(w), w)$ if all $u(t_i)(w)$ are defined and undefined otherwise; $f^\mathcal{U}$ is the image of $f$ by the attribute meaning function $m^A_\mathcal{U}$;

• If $t_1$ is a term of sort $s_1, \ldots, t_n$ a term of sort $s_n$ and $f : s_1\ldots s_n \to \text{bool}$ is a pointcut signature, then $u(f(t_1, t_2, \ldots, t_n))(w) = f^\mathcal{U}(u(t_1)(w), u(t_2)(w), \ldots, u(t_n)(w))$ if all $u(t_i)(w)$ are defined and undefined otherwise; $f^\mathcal{U}$ is the image of $f$ by the pointcut meaning function $m^P_\mathcal{U}$;

This term interpretation is obtained by using the $\Sigma$-homomorphism definition 5.25. Note that an action and a pointcut interpretations are independent of the state in which the component is in while an attribute interpretation is state dependent.

### 5.5.1.2.3 Satisfaction Relation

Given a structure $M = \langle W, \mathcal{U}, r, l, f_j, pr \rangle$, the set $T_\Sigma(X)$ of all terms over a signature $\Sigma$ and a $S$-sorted set of variables $X$, and a valuation $u : T_\Sigma(X) \to \mathcal{U}$, the satisfaction relation $\models$ is defined as follows. We write $M, w, u \models \varphi$ or $w \models \varphi$ to mean that the formula $\varphi$ is satisfied in $M$ at the state $w$ under the valuation $u$. The satisfaction relation $\models$ has the same meaning of one in the logic $L_A$ (defined in chapter 4). We can also use the relation $\models_a$ used in $L_A$ for actions. $\mathcal{F}$ is the set of the well-formed formulae relative to a $\Sigma$. Let $\varphi \in \mathcal{F}$. $u(\varphi)$ is the image of the formula $\varphi$ by the homomorphism $u$. We have:

$$w \models \varphi \Rightarrow w \models u(\varphi)$$
5.5.1.2.4 Satisfiability and Validity

Definition 5.29. Let $\phi \in \mathcal{F}$ be a formula. If there exists a structure $M = \langle W, \mathcal{U}, r, l, f_j, pr \rangle$, a state $w \in W$, and a valuation $u : T_\Sigma(X) \rightarrow \mathcal{U}$ such that $M, w, u \models \phi$, we say that $\phi$ is satisfiable.

Definition 5.30. Let $M = \langle W, \mathcal{U}, r, l, f_j, pr \rangle$ be a structure, and $\phi \in \mathcal{F}$. $\phi$ is valid w.r.t $M$ or $M$ satisfies $\phi$ or $M$ is a model of $\phi$ denoted by $M \models \phi$ if $M, w, u \models \phi$ for all $w \in W$ and for all valuation $u : T_\Sigma(X) \rightarrow \mathcal{U}$. If $\Phi \subseteq \mathcal{F}$, we write $M \models \Phi$ if $M \models \phi$, $\forall \phi \in \Phi$ and we say that $M$ satisfies $\Phi$ or that $M$ is a model of $\Phi$. We denote Mod$\Phi$ the class of all models of $\Phi$.

Definition 5.31. Let $\phi \in \mathcal{F}$. $\phi$ is valid (denoted by $\models \phi$) if $M \models \phi$ for all structures $M$.

Definition 5.32. Model of a Specification.

Let SPEC be a specification consisting of a signature $\Sigma$, a set of variables VAR, and a set $\Phi = AX \cup PAX \cup AAX \cup PrAX$ of axioms, where $AX$ is a finite set of axioms (defined on the set of data terms), $PAX$ is a finite set of point-cut axioms, $AAX$ is a finite set of advice axioms, and $PrAX$ is finite set of prescription axioms. Note that SPEC could be a Class, or an Aspect specification. A model of SPEC is a structure $M = \langle W, \mathcal{U}, r, l, f_j, pr \rangle$ such that $M \models \Phi$. Mod$\Phi$ is called the loose semantics of SPEC.

Definition 5.33. As in [107], we assume that Mod$\Phi$ is not empty, that is to say $\Phi$ is consistent. Otherwise, when Mod$\Phi = \emptyset$, the specification SPEC is incorrect.

5.5.1.3 Semantics of ADT Specifications

Let $\Sigma = \langle S, \leq, AC \rangle$ be an ADT signature. The elements of the set AC do not have any effect on the states in a system. In other words, we cannot perform an action of AC and say that we move from a state to a state (such as in a case of class or aspect component). ADTs are simply used by class and aspect components. Thus, in this context, the interpretation structures we consider here is of the form $M = \langle \mathcal{U}, l \rangle$ where $\mathcal{U}$ is an ADT $\Sigma$-algebra and $l$ the above defined function (see section 5.5.1.2.1). All definitions in the section 5.5.1.2 are valid here by removing the concept of state. The set $\Phi$ is equal to $AX$. 
5.5.2 Semantics of Module Specifications

5.5.2.1 Interpretation of Specification Morphisms

**Definition 5.34. Interpretation of ADT Signature Morphisms.**

Let $\Sigma_1 = \langle S_1, \leq_1, AC_1 \rangle$ and $\Sigma_2 = \langle S_2, \leq_2, AC_2 \rangle$ be two ADT signatures, and $\sigma = (\sigma_1, \sigma_2) : \Sigma_1 \to \Sigma_2$ a signature morphism where $\sigma_1 : S_1 \to S_2$ and $\sigma_2 : AC_1 \to AC_2$. Let $\mathcal{A}_1 = \langle |\mathcal{A}_1|, m^A_{\mathcal{A}_1} \rangle$ be a $\Sigma_1$-Algebra and $\mathcal{A}_2 = \langle |\mathcal{A}_2|, m^A_{\mathcal{A}_2} \rangle$ be a $\Sigma_2$-Algebra. Let $\sigma_1^{\mathcal{A}} : |\mathcal{A}_1| \to |\mathcal{A}_2|$ be a map such that:

- $\forall s \in S_1, \sigma_1^{\mathcal{A}}(|\mathcal{A}_1|_s) \subseteq |\mathcal{A}_2|_{\sigma_1(s)}$

- $\forall s, s' \in S_1, s \leq_1 s' \Rightarrow |\mathcal{A}_2|_{\sigma_1(s)} \subseteq |\mathcal{A}_2|_{\sigma_1(s')}$

Let $AC^{\mathcal{A}_1} = \{f^{\mathcal{A}_1}, f \in AC_1\}$ and $AC^{\mathcal{A}_2} = \{f^{\mathcal{A}_2}, f \in AC_2\}$. Let $\sigma_2^{\mathcal{A}} : AC^{\mathcal{A}_1} \to AC^{\mathcal{A}_2}$ be a map such that:

- $\sigma_2^{\mathcal{A}}(f^{\mathcal{A}_1}) = (\sigma_2(f))^{\mathcal{A}_2}$

We call $\sigma^{\mathcal{A}} = (\sigma_1^{\mathcal{A}}, \sigma_2^{\mathcal{A}}) : \mathcal{A}_1 \to \mathcal{A}_2$ a meaning ADT signature morphism of the signature morphism $\sigma = (\sigma_1, \sigma_2)$.

Recall that if $f : \omega \to s \in AC_1$, then $f^{\mathcal{A}_1} : |\mathcal{A}_1|_\omega \to |\mathcal{A}_1|_s$ and $\sigma_2(f) : \sigma_1(\omega) \to \sigma_1(s)$, and $(\sigma_2(f))^{\mathcal{A}_2} : |\mathcal{A}_1|_{\sigma_1(\omega)} \to |\mathcal{A}_1|_{\sigma_1(s)}$.

**Definition 5.35. Interpretation of Class Signature Morphisms.**

Given the Class signatures $\Sigma_1 = \langle S_1, \leq_1, AC_1, ST_1 \rangle$ and $\Sigma_2 = \langle S_2, \leq_2, AC_2, ST_2 \rangle$, let $\sigma = (\sigma_1, \sigma_2, \sigma_3) : \Sigma_1 \to \Sigma_2$ be a Class signature morphism, where $\sigma_1 : S_1 \to S_2$, $\sigma_2 : AC_1 \to AC_2$, and $\sigma_3 : ST_1 \to ST_2$. Let $\mathcal{A}_1 = \langle |\mathcal{A}_1|, m^A_{\mathcal{A}_1}, m^S_{\mathcal{A}_1} \rangle$ be a $\Sigma_1$-Algebra and $\mathcal{A}_2 = \langle |\mathcal{A}_2|, m^A_{\mathcal{A}_2}, m^S_{\mathcal{A}_2} \rangle$ be a $\Sigma_2$-Algebra. Let $ST^{\mathcal{A}_1} = \{f^{\mathcal{A}_1}, f \in ST_1\}$ and $ST^{\mathcal{A}_2} = \{f^{\mathcal{A}_2}, f \in ST_2\}$. Let $\sigma_3^{\mathcal{A}} : ST^{\mathcal{A}_1} \to ST^{\mathcal{A}_2}$ be a map such that:

- $\sigma_3^{\mathcal{A}}(f^{\mathcal{A}_1}) = (\sigma_3(f))^{\mathcal{A}_2}$

The map $\sigma^{\mathcal{A}} = (\sigma_1^{\mathcal{A}}, \sigma_2^{\mathcal{A}}, \sigma_3^{\mathcal{A}}) : \mathcal{A}_1 \to \mathcal{A}_2$ where $(\sigma_1^{\mathcal{A}}, \sigma_2^{\mathcal{A}})$ is a meaning ADT signature morphism, is called a meaning Class signature morphism of the Class signature morphism $\sigma = (\sigma_1, \sigma_2, \sigma_3)$.

**Definition 5.36. Interpretation of Aspect Signature Morphisms.**

Given Aspect signatures $\Sigma_1 = \langle S_1, \leq_1, AC_1, ST_1, PC_1 \rangle$ and $\Sigma_2 = \langle S_2, \leq_2$
, AC₂, ST₂, PC₂), let \( \sigma = (\sigma_1, \sigma_2, \sigma_3, \sigma_4) : \Sigma_1 \rightarrow \Sigma_2 \) be an Aspect signature morphism, where \( \sigma_1 : S_1 \rightarrow S_2 \), \( \sigma_2 : AC_1 \rightarrow AC_2 \), \( \sigma_3 : ST_1 \rightarrow ST_2 \), and \( \sigma_4 : PC_1 \rightarrow PC_2 \). Let \( \mathcal{A}_1 = \langle |\mathcal{A}_1|, m^A_{\mathcal{A}_1}, m^S_{\mathcal{A}_1}, m^P_{\mathcal{A}_1} \rangle \) be a \( \Sigma_1 \)-Algebra and \( \mathcal{A}_2 = \langle |\mathcal{A}_2|, m^A_{\mathcal{A}_2}, m^S_{\mathcal{A}_2}, m^P_{\mathcal{A}_2} \rangle \) be a \( \Sigma_2 \)-Algebra. Let \( PC_{\mathcal{A}_1} = \{pc_{\mathcal{A}_1}, pc \in PC_1 \} \) and \( PC_{\mathcal{A}_2} = \{pc_{\mathcal{A}_2}, pc \in PC_2 \} \). Let \( \sigma^A_4 : PC_{\mathcal{A}_1} \rightarrow PC_{\mathcal{A}_2} \) be a map such that:

\[
\bullet \quad \sigma^A_4(pc_{\mathcal{A}_1}) = (\sigma_4(pc))_{\mathcal{A}_2}
\]

The map \( \sigma^A = (\sigma_1^A, \sigma_2^A, \sigma_3^A, \sigma_4^A) : \mathcal{A}_1 \rightarrow \mathcal{A}_2 \) where \( (\sigma_1^A, \sigma_2^A, \sigma_3^A) \) is a meaning Class signature morphism, is called a meaning Aspect signature morphism of the Aspect signature morphism \( \sigma = (\sigma_1, \sigma_2, \sigma_3, \sigma_4) \).

**Definition 5.37. Interpretation of Aspect-Class Signature Morphisms.**

Let \( \Sigma_1 = \langle S_1, \leq_1, AC_1, ST_1, PC_1 \rangle \) be an aspect signature and \( \Sigma_2 = \langle S_2, \leq_2, AC_2, ST_2 \rangle \) a class signature. Let \( \sigma = (\sigma_1, \sigma_2, \sigma_3, \sigma_4) : \Sigma_1 \rightarrow \Sigma_2 \) be an Aspect-Class signature morphism, where \( \sigma_1 : S_1 \rightarrow S_2 \), \( \sigma_2 : AC_1 \rightarrow AC_2 \), \( \sigma_3 : ST_1 \rightarrow ST_2 \), and \( \sigma_4 : PC_1 \rightarrow AC_2 \). Let \( \mathcal{A}_1 = \langle |\mathcal{A}_1|, m^A_{\mathcal{A}_1}, m^S_{\mathcal{A}_1}, m^P_{\mathcal{A}_1} \rangle \) be a \( \Sigma_1 \)-Algebra and \( \mathcal{A}_2 = \langle |\mathcal{A}_2|, m^A_{\mathcal{A}_2}, m^S_{\mathcal{A}_2} \rangle \) be a \( \Sigma_2 \)-Algebra. Let \( \sigma^A_4 : PC_{\mathcal{A}_1} \rightarrow AC_{\mathcal{A}_2} \) be a map such that:

\[
\bullet \quad \sigma^A_4(pc_{\mathcal{A}_1}) = (\sigma_4(pc))_{\mathcal{A}_2}
\]

The map \( \sigma^A = (\sigma_1^A, \sigma_2^A, \sigma_3^A, \sigma_4^A) : \mathcal{A}_1 \rightarrow \mathcal{A}_2 \) where \( (\sigma_1^A, \sigma_2^A, \sigma_3^A) \) is a meaning Class signature morphism, is called a meaning Aspect-Class signature morphism of the Aspect-Class signature morphism \( \sigma = (\sigma_1, \sigma_2, \sigma_3, \sigma_4) \).

**Theorem 5.11.** Let \( SPEC_1 \) and \( SPEC_2 \) be two algebraic specifications and \( \sigma : SPEC_1 \rightarrow SPEC_2 \) a specification morphism. Let \( M_1 \) and \( M_2 \) be respectively a model of \( SPEC_1 \) and \( SPEC_2 \). Let \( \Phi_1 \) be the axiom set of \( SPEC_1 \), \( \Phi_2 \) be the axiom set of \( SPEC_2 \) and \( \varphi \in \Phi_1 \). We have for all valuations \( u_1 \) related to \( M_1 \) and for all valuations related to \( M_2 \):

\[
M_1 \models u_1(\varphi) \Rightarrow M_2 \models (u_2 \circ \sigma)(\varphi)
\]

**Proof.** We make the proof in the case where \( SPEC_1 \) and \( SPEC_2 \) are class or aspect algebraic specifications. The proof of the case where \( SPEC_1 \) and \( SPEC_2 \) are ADT algebraic specifications is similar by removing the notion of state. \( M_1 \models \varphi \Rightarrow M_1, \omega_1, u_1 \models \varphi, \forall \omega_1, u_1 \). That implies \( M_2, \omega_2, u_2 \models \sigma(\varphi), \forall \omega_2, u_2 \) because, \( \varphi \in \Phi_1 \Rightarrow \sigma(\varphi) \in \Phi_2 \). That implies \( M_2, \omega_2, u_2 \models u_2(\sigma(\varphi)), \forall \omega_2, u_2 \) (because \( u_2 \) is a homomorphism). Whence, \( M_1 \models u_1(\varphi) \Rightarrow M_2 \models (u_2 \circ \sigma)(\varphi) \). \( \square \)
**Definition 5.38. Interpretation of Class or Aspect Specification Morphisms.**

Let $\text{SPEC}_1$ and $\text{SPEC}_2$ be two (Class or Aspect) algebraic specifications. Let $M_1 = \langle W_1, \mathcal{V}_1, r_1, l_1, f_1, pr_1 \rangle$ be a model of $\text{SPEC}_1$ and $M_2 = \langle W_2, \mathcal{V}_2, r_2, l_2, f_2, pr_2 \rangle$ a model of $\text{SPEC}_2$. Let $\sigma : \text{SPEC}_1 \to \text{SPEC}_2$ a specification morphism. A meaning signature morphism $\sigma^{\text{sig}} : \mathcal{V}_1 \to \mathcal{V}_2$ is a meaning specification morphism from $M_1$ to $M_2$.

- If $\text{SPEC}_1$ and $\text{SPEC}_2$ are class specifications, we talk about a meaning class specification morphism.
- If $\text{SPEC}_1$ and $\text{SPEC}_2$ are aspect specifications, we talk about a meaning aspect specification morphism.
- If $\text{SPEC}_1$ is an aspect specification and $\text{SPEC}_2$ is class specification, we talk about a meaning aspect-class specification morphism.

**Definition 5.39. Interpretation of ADT Specification Morphisms.**

Let $\text{SPEC}_1$ and $\text{SPEC}_2$ be two ADT specifications. Let $M_1 = \langle \mathcal{V}_1, l_1 \rangle$ be a model of $\text{SPEC}_1$ and $M_2 = \langle \mathcal{V}_2, l_2 \rangle$ a model of $\text{SPEC}_2$. Let $\sigma : \text{SPEC}_1 \to \text{SPEC}_2$ an ADT specification morphism. A meaning signature morphism $\sigma^{\text{sig}} : \mathcal{V}_1 \to \mathcal{V}_2$ is a meaning specification morphism from $M_1$ to $M_2$.

**5.5.2.2 Semantics of Module Specifications**

Recall that a module specification $MOD$ is defined by the following commutative diagram:

$$
\begin{array}{c}
\text{PAR} \xrightarrow{e} \text{EXP} \\
\downarrow i \quad \downarrow v \\
\text{IMP} \xrightarrow{s} \text{BOD}
\end{array}
$$

**Figure 5.9: Module specification**

$MOD = \langle \text{PAR}, \text{EXP}, \text{IMP}, \text{BOD}, e, s, i, v \rangle$.

**Definition 5.40.** Let $MOD = \langle \text{PAR}, \text{EXP}, \text{IMP}, \text{BOD}, e, s, i, v \rangle$ be a module specification. Let $M_P, M_E, M_I, M_B$ be respectively a model of $\text{PAR}, \text{EXP}, \text{IMP}, \text{BOD}$. Let
\[ e^{\text{id}} : M_P \rightarrow M_E, \ s^{\text{id}} : M_I \rightarrow M_B, \ i^{\text{id}} : M_P \rightarrow M_I \text{ and } v^{\text{id}} : M_E \rightarrow M_B \] be respectively a meaning specification morphism of the specification morphisms \( e, s, i \) and \( v \) such that the following diagram 5.10 commutes.

![Diagram](image)

**Figure 5.10: Model of a Module specification**

The tuple \( M = \langle M_P, M_E, M_I, M_B, e^{\text{id}}, s^{\text{id}}, i^{\text{id}}, v^{\text{id}} \rangle \) is a model of the module specification \( MOD \).

- The set of all such tuples constitutes the loose semantics of \( MOD \).

- If \( MOD \) is an ADT module specification, \( M_P, M_E, M_I, M_B \) are ADT models of \( PAR, EXP, IMP, BOD \), \( e^{\text{id}}, i^{\text{id}}, v^{\text{id}}, s^{\text{id}} \) are ADT meaning morphisms, and \( M \) is called an ADT model of \( MOD \).

- If \( MOD \) is a Class module specification, \( M_P, M_E, M_I, M_B \) are Class models of \( PAR, EXP, IMP, BOD \), \( e^{\text{id}}, i^{\text{id}}, v^{\text{id}}, s^{\text{id}} \) are Class meaning morphisms, and \( M \) is called a Class model of \( MOD \).

- If \( MOD \) is an Aspect module specification, \( M_P, M_E, M_I, M_B \) are Aspect models of \( PAR, EXP, IMP, BOD \), \( e^{\text{id}}, i^{\text{id}}, v^{\text{id}}, s^{\text{id}} \) are Aspect meaning morphisms, and \( M \) is called an Aspect model of \( MOD \).

Let \( \text{Mod}_{\text{Spec}} \) the set of all models of all algebraic specifications, and all meaning specification morphisms of a system. Likewise, let \( \text{Mod}_{\text{Mod}} \) the set of all models of all module specifications, and all meaning module morphisms of a system.

**Theorem 5.12.** \( \text{Mod}_{\text{Spec}} \) and \( \text{Mod}_{\text{Mod}} \) form respectively a category.

**Proof.** It is easy to verify that \( \text{Mod}_{\text{Spec}} \) and \( \text{Mod}_{\text{Mod}} \) satisfy the constraints of the definition of category (2.3) of chapter 2. \( \square \)

We don’t prove that \( \text{MOD}_{\text{Mod}} \) is finitely co-complete (even we don’t know that is). However, we have this theorem:
Theorem 5.13. Any finite diagram of $\text{CAT}_{\text{Mod}}$ has a semantics (or a meaning) in the semantics domain $\text{MOD}_{\text{Mod}}$.

Proof. Recall that $\text{CAT}_{\text{Mod}}$ is the set of all ADT, Class and Aspect module specifications and their different morphisms of a system. Since $\text{CAT}_{\text{Mod}}$ is finitely co-complete category (see section 5.4.2), any finite diagram of $\text{CAT}_{\text{Mod}}$ has a colimit which is a module specification. This module specification has a semantics. \qed

5.6 Conclusion

In this chapter, we defined the syntaxes and the semantics of our approach. This framework is based on category theory and algebraic specification due to their formality, their modularity benefits, and their high level of abstraction. Large complex software systems are composed of many software components. These components could be aspect components or class components. Construction and maintenance of such systems require a clear understanding of the dependencies between these components. Software components represent the main functional pieces of an application, while dependencies describe their interconnection relationships. Category theory is a useful tool to clearly specify these components and dependencies. Each component is represented by a module specification and the relationships between the components are expressed by means of morphisms.

We choose to use a loose semantics because it is natural and simple. This semantics uses algebras and the logic $L_A$ (defined in chapter 4). A semantics of an algebraic specification or a module specification is defined as a set of deontic models that validate the axioms of this algebraic specification or this module specification. These deontic models differ from more conventional ones, for example one in [24, 25] where semantics is defined by an interface semantics (constituted of categories of parameter, import, and export with two functors), a construction semantics (defined by a free functor from the category of import models to the category of body models), and a behavior semantics represented by a functor between the categories of import and export models. We think that this semantics is too complex. Our model uses the semantics model of the logic $L_A$ and has advantages that allows us to formalize information like error recovery through corrective action or sanctions if desired. More, it integrates our prevention mechanism which will allows to avoid undesirable behaviors in aspect-oriented systems. In this structure, we can distinguish between description and prescription of behavior. We are aware that formal semantics of the aspect concepts are lacking in AOSD.
CHAPTER 6

MODULE INTERCONNECTION: WEAVING ALGORITHM

6.1 Introduction

As said in the chapter 5, a system is decomposed into aspect components and class components. Each component is represented by a module specification. These module specifications are related by morphisms defining the relations between these modules. After the system has been decomposed into aspect and class components, and the module specifications and morphisms have been specified, then we have a diagram for this system in which the nodes are module specifications and the fleches are morphisms. This diagram is a modules net showing the way modules interact with each other or how they are tight together. Module interconnections form the architectural structure of a modular system. Four types of interconnection have been defined in [25] and are used in [110]:

1. Use or Client/server relationship: relationship in which one module uses resources of the second one.

2. Share relationship: relationship in which two modules share some elements. This type of relationship can be used to describe the synchronous communication.

3. Actualization relationship: relationship between a generic module and an actual module.

4. Extension or adding relationship: some elements are added in a module.

For each of the four relationships, construction operators have been defined by Ehrig and Mahr [25] to realize the interconnection associated.

We extend the four types of interconnection by adding a fifth kind, the weaving relation. The process of weaving an aspect to a set of base objects (classes) consists in assembling these entities together to produce the final application extended with the behaviors defined in the aspects. We define an algorithm that takes an aspect module and a class module as inputs and outputs an augmented module. Our algorithm represents the operator of the weaving relation and it is based on the notion of colimits and the union operator.

The colimit of an interconnection diagram is a new object, called the colimiting object, together with a family of mappings, one from each object in the diagram onto the
colimiting object. Since each mapping expresses how the internal structure of its source object is mapped onto that of its destination object, the colimit expresses the merge of all the objects in the interconnection diagram. Furthermore, the colimit respects the mappings in the diagram: The intuition here is that the image of each object in the colimit is the same, no matter which path through the mappings in the diagram you follow. By definition, the colimit is also minimal; it merges the objects in the diagram without adding anything essentially new. Hence, the use of the notion of colimit is an advantage of our weaving formalism.

The remainder of this chapter is organized as follows. The section 6.2 describes in details aspect-class relationship. Section 6.3 presents the description of the interconnection mechanisms defined in [25]. Section 6.4 presents our weaving algorithm. In section 6.5 we conclude this chapter.

6.2 Aspect-Class Relationship

6.2.1 Weaving

An Aspect-Oriented Modeling (AOM) design consists of one or more aspects and a primary model. Aspect models describe behavior that cross-cuts the primary model (a set of classes in object-oriented sense). An aspect component embodies a crosscutting concern. It is a component that provides as services pieces of advice code. An aspect can be seen as a server, whereas class can be viewed as a client. The primary components expose a set of joinpoints on which aspect components can be woven. The interface of each advice is defined in the aspect interface. An aspect-class binding is a communication channel between a client interface and a server interface. A client interface uses operations provided by a server interface. In order to obtain an integrated design view, aspect and primary models must be composed. The process of weaving an aspect to a set of base objects consists in assembling these entities together to produce the final application extended with the behaviors defined in the aspects. We distinguish static (compile time) weaver from dynamic (runtime) weaver. Pointcut expressions are used to weave an aspect component on a set of components. When an interface of a component matches the expression, the interface is bound to the aspect component. Note that an aspect can be woven into an aspect. A weaver takes as inputs an aspect component and the primary component (base system) and outputs an augmented component. This process is represented in the figure 6.1 and can be iterated many times possible. Also, aspect can be
removed from an augmented component.

Figure 6.1: The Weaving Process

At the implementation level, we can observe the following in the weaving process (statically):

**Before the weaving**

```java
public class HelloWorld {
    public static void main (String[] args) {
        new HelloWorld().sayHello();
    }
    public void sayHello () {
        System.out.println("Hello World!");
    }
}
```

**Figure 6.2: Hello word class**

```java
public aspect World {
    pointcut greeting():
        execution(*HelloWorld.sayHello(.));
    after(): greeting(){
        System.out.println("Hello you too...");
    }
}
```

**Figure 6.3: Hello word aspect**
After the weaving

```java
public class HelloWorld {
    public static void main (String[] args)
    {
        new HelloWorld().sayHello();
        System.out.println("Hello, you too...");
    }
    public void sayHello ()
    {
        system.out.println("Hello, World!");
    }
}
```

Figure 6.4: Hello word augmented system

We want to identify or define categorical operators that can allow to compute this augmented system at the specification level.

6.2.2 Relationship between Aspect and Class Modules

As John Donne once famously said: "no man is an island, entire of itself". The same situation exists in our programs - no object or aspect stands alone. Indeed, without relationships, most objects and aspect are meaningless. Aspect-component relation is effectively a binding of an aspect to a component. Connection between aspect and class modules could be described as "an aspectual module A affects a class module C by means of a crosscutting relationship". The crosscutting relationship relates aspects to base components. It may include a list of bindings. A UML representation of the relation between an aspect and a class is described in the paper [101]. Figure 5.4 of chapter 5 taking from the paper [101] is an example of the connection between aspects and classes. Aspects are connected to classes using unidirectional links. There are two tables in this example. Each table characterizes the relationship that exists between an aspect A and a class C. The top table describes the relationship between the aspect Log and the class Invoice while the bottom table describes the relationship between the aspect Verification and the class Invoice. Our objective is to characterize these relationships in the Ehrig and Mahr formalism [25]. Each line in the tables defines a connection link between an aspect and a class. The columns define the necessary information in order to characterize
these relationships.

In the paper [52], connection between an aspect and a class is described by a pointcut relation that allows the designer to link some aspect-methods (belonging to aspect) to some points of the base-class. It is an oriented association from an aspect towards a classifier such as a class (or a group if the aspect works on heterogeneous or distributed objects). The association is stereotyped with \( \ll \text{pointcut} \rr \) and the roles have special semantics since they are used to tell which methods of the client-class are extended and by which aspect-methods.

Figure 6.5: The pointcut association: relating aspects to classes

Figure 6.5 (taking from [52]) shows the relation between an aspect A and a class C:

- a pointcut relation \( p \) must go from an aspect A to a classifier C (that can be a group);
- role \( r_1 \) is the name of an aspect-method defined in A (must be an around, before, or after method) that is applied at each base-program point denoted by role \( r_2 \);
- role \( r_2 \) defines a base-program crosscut (a set of points); it is of the form \( T[expr] \) where \( T \) is the type of the joinpoint and \( expr \) is an optional expression that specifies (acts like a filter for) the set of points;
- cardinality \( c_1 \) is the number of aspect instances of A that can be in relation with one instance of C;
- cardinality \( c_2 \) is the number of instances of C that can be in relation with one instance of A;
- as any UML model element, the pointcut relation can be tagged (tag) to express extra semantics that can be used when implementing the model towards a concrete platform.
The next section will describe the four interconnection operations defined in [25]. We will use some of these operators to define our weaving algorithm.

6.3 Interconnection of Modules

As said in [25], module interconnections define the way modules interact with each other or how they are tight together. Module interconnections form the architectural structure of a modular system. In chapter 2, we define some categorical basic constructions that allow composing two or more objects or software components and then getting a larger object or component that represents the composed objects. Four types of interconnection have been defined in [25] and are clearly expressed in [110]:

1. Use or Client/server relationship: a relationship in which one module uses resources of the second one.

2. Share relationship: a relationship in which two modules share some elements. This type of relationship can be used to describe the synchronous communication.

3. Actualization relationship: relationship between a generic module and an actual module.

4. Extension or adding relationship: some elements are added in a module.

For each of the four relationships, construction operators have been defined by Ehrig and Mahr [25] to realize the interconnection associated. We have for the:

1. Use or client/server relationship: composition, partial composition (generalization of composition), product construction operators.

2. Share relationship: union construction operator; it is in fact a pushout of modules.


After connecting all module specifications of the system, it is the time to build the module representing the description of the global system. In this section, we describe these construction operators that use some universal constructions defined in chapter 2. These operators constitute interconnection mechanisms aiming to combine and extend module specifications and get a modular system. Such operators have module specifications as
arguments and result. Ehrig and Mahr [25] show that the interconnection of correct module specifications is again a correct one that the semantics of the resulting module specification can be defined in terms of semantics of the parts.

6.3.1 Modules Composition

In this section, we describe the composition operator. This operator takes two module specifications as inputs and returns a module specification. The import interface of one module specification is matched to the export interface of the second module. Matching of two specifications $SPEC_1$ and $SPEC_2$ in this context means the existence of a specification morphism $h : SPEC_1 \rightarrow SPEC_2$.

**Definition 6.1. Module Composition.**

The composition of two module specifications $MOD_j = \langle PAR_j, EXP_j, IMP_j, BOD_j, e_j, s_j, i_j, v_j \rangle$ for $j = 1, 2$ via a pair $h = \langle h_1, h_2 \rangle$ of specification morphisms $h_1 : IMP_1 \rightarrow EXP_2$ and $h_2 : PAR_1 \rightarrow PAR_2$ satisfying $e_2 \circ h_2 = h_1 \circ i_1$ ($\circ$ is a function composition law) is the module specification $MOD_3 = \langle PAR_3, EXP_3, IMP_3, BOD_3, e_3, s_3, i_3, v_3 \rangle$ given by the outer square in the figure 6.6: with $EXP_3 = EXP_1$, $IMP_3 = IMP_2$, $PAR_3 = PAR_1$ and $BOD_3 = BOD_1 +_{IMP_1} BOD_2$ is the pushout of $BOD_1$ and $BOD_2$ via $IMP_1$ in square.
(4), \( e_3 = e_1, s_3 = b_2 \circ s_2, i_3 = i_2 \circ h_2 \), and \( \nu_3 = b_1 \circ \nu_1 \) where \( b_1 \) and \( b_2 \) are defined by the pushout square (4). \( MOD_3 \) is noted \( MOD_3 = MOD_1 \circ h \ MOD_2 \).

Remark 6.1. • The existence of \( h_1 \) means that at least the requirements formulated in \( IMP_1 \) are satisfied by \( EXP_2 \) but \( EXP_2 \) may provide additional features which are not used by \( IMP_1 \). The existence of \( h_2 \) is motivated by the fact that the parameter parts of the modules are also intended to be parameters of the result module. \( e_2 \circ h_2 = h_1 \circ i_1 \) means that the matching made by \( h_2 \) is compatible with the matching made by \( h_1 \).

• In this definition, the module specification \( MOD_1 \) uses the module specification \( MOD_2 \). \( h_1 \) matches the import interface \( IMP_1 \) of \( MOD_1 \) to the export interface of \( MOD_2 \) and \( h_2 \) matches the parameter part \( PAR_1 \) of \( MOD_1 \) to the parameter part \( PAR_2 \) of \( MOD_2 \). The remaining import interface \( IMP_2 \) becomes the import interface of \( MOD_3 \) and the remaining export interface \( EXP_1 \) becomes the export interface of \( MOD_3 \). The common body \( BOD_3 \) is the body \( BOD_1 \) where \( IMP_1 \) is replaced by \( BOD_2 \). Note that \( EXP_2 \)-operations are not going to be exported in the composition, unless they are already in \( EXP_1 \).

• The composition \( MOD_3 \) is again a module specification because from the fact the four internal diagrams commute, the outer diagram commutes.

• Do not confound composition of modules with the composition of morphisms.

6.3.2 Product of Modules

The product operation is an alternative to the composition operation where the export of the product - in contrast to the export of the composition - is the union of the export of both module specifications. Moreover, the resulting parameter is no longer that of the upper module \( MOD_1 \) but that of the lower one \( MOD_2 \). This allows avoiding the second part \( h_2 \) of the interface passing morphism \( h = \{ h_1, h_2 \} \) which is used in the composition operation.

Definition 6.2. Module Product.
The product of two module specifications \( MOD_j = \{ PAR_j, EXP_j, IMP_j, BOD_j, e_j, s_j, i_j, \nu_j \} \) for \( j = 1, 2 \) via a specification morphisms
$h_1 : IMP_1 \rightarrow EXP_2$, called interface passing morphism, is the module specification $MOD_3 = \langle PAR_3, EXP_3, IMP_3, BOD_3, e_3, s_3, i_3, v_3 \rangle$ given by the diagram $(PAR_2, EXP_3, IMP_2, BOD_3)$ in the figure 6.7:

![Diagram](image)

Figure 6.7: Modules Product

with $EXP_3 = EXP_1 +_{PAR_1} EXP_2$ is the pushout of $EXP_1$ and $EXP_2$ via $PAR_1$ in the diagram $\langle PAR_1, EXP_1, EXP_3, EXP_2 \rangle$, $IMP_3 = IMP_2$, $PAR_3 = PAR_2$ and $BOD_3 = BOD_1 +_{IMP_1} BOD_2$ is the pushout of $BOD_1$ and $BOD_2$ via $IMP_1$ in the diagram $\langle IMP_1, BOD_1, BOD_3, BOD_2 \rangle$, $e_3 = x_2 \circ e_2$, $s_3 = b_2 \circ s_2$, $i_3 = i_2$, and $v_3$ the unique induced morphism out of the pushout $EXP_3$ making the diagram above commutative where $b_1$ and $b_2$ are defined by the pushout square $\langle IMP_1, BOD_1, BOD_3, BOD_2 \rangle$. $MOD_3$ is noted $MOD_3 = MOD_1 *_{h_1} MOD_2$.

Remark 6.2. Do not confound this notion of product of modules with the product of objects in a category.

6.3.3 Union of Modules

We describe in this section the Union mechanism of two module specifications a long with shared submodule specification. We first give the definition of a submodule.

**Definition 6.3.** Submodule specification.

A module specification $MOD_0 = \langle PAR_0, EXP_0, IMP_0, BOD_0, e_0, s_0, i_0, v_0 \rangle$
is a submodule specification of a module specification \( MOD_1 = \langle PAR_1, EXP_1, IMP_1, BOD_1, e_1, s_1, i_1, v_1 \rangle \) if there exist four inclusion specification morphisms \( m_P : PAR_0 \to PAR_1, m_I : IMP_0 \to IMP_1, m_E : EXP_0 \to EXP_1, m_B : BOD_0 \to BOD_1 \) such that the following diagrams commute:

\[
\begin{array}{ccc}
PAR_0 & \xrightarrow{e_0} & EXP_0 & \xrightarrow{v_0} & BOD_0 \\
\downarrow m_P & & \downarrow m_E & & \downarrow m_B \\
PAR_1 & \xrightarrow{e_1} & EXP_1 & \xrightarrow{v_1} & BOD_1 \\
\end{array}
\]

\[
\begin{array}{ccc}
PAR_0 & \xrightarrow{i_0} & IMP_0 & \xrightarrow{s_0} & BOD_0 \\
\downarrow m_P & & \downarrow m_I & & \downarrow m_B \\
PAR_1 & \xrightarrow{i_1} & IMP_1 & \xrightarrow{s_1} & BOD_1 \\
\end{array}
\]

**Remark 6.3.** A submodule is a module such that the inclusions morphisms \( m_P, m_I, m_E, m_B \) are module morphisms.

Union of module specifications is defined componentwise. Union operation identifies part common to the module specifications to be united. These common parts must be declared. The connection between the shared submodule with other module specifications is given by a pair of module specification morphisms. The union construction turns out to be a pushout of these morphisms in the category of module specifications and module specification morphisms.

**Definition 6.4. Union of Modules with shared submodule.**

Given module specifications \( MOD_j = \langle PAR_j, EXP_j, IMP_j, BOD_j, e_j, s_j, i_j, v_j \rangle \) for \( j = 0, 1, 2 \), and module specification morphisms \( f_1 : MOD_0 \to MOD_1 \) and \( f_2 : MOD_0 \to MOD_2 \) such that \( f_1 = \langle f_{1P}, f_{1E}, f_{1I}, f_{1B} \rangle \) and \( f_2 = \langle f_{2P}, f_{2E}, f_{2I}, f_{2B} \rangle \) are inclusions, the union \( MOD_3 = \langle PAR_3, EXP_3, IMP_3, BOD_3, e_3, s_3, i_3, v_3 \rangle \) of \( MOD_1 \) and \( MOD_2 \) with the shared submodule \( MOD_0 \) is defined as follows:

- \( PAR_3 = PAR_1 +_{PAR_0} PAR_2 \)
- \( EXP_3 = EXP_1 +_{EXP_0} EXP_2 \)
- \( IMP_3 = IMP_1 +_{IMP_0} IMP_2 \)
• $BOD_3 = BOD_1 + BOD_0 BOD_2$

• The specification morphisms $e_3, s_3, i_3$ and $v_3$ of $MOD_3$ are uniquely defined by the pushout properties of $PAR_3$ (2 times), $IMP_3$ and $EXP_3$ in the following figure 6.8:

![Diagram](https://via.placeholder.com/150)

Figure 6.8: Union of two Module Specifications

**Calculus:**
The definition of $e_3, s_3, i_3$, and $v_3$ uses the fact that $f_1$ and $f_2$ are module specification morphisms such that all sub-diagrams of the diagram 6.8 except one consisting of $e_3, s_3, i_3$, and $v_3$ are already commutative.
$PAR_3$ and the two morphisms $g_{1p}, g_{2p}$ are pushout of $f_{1p}, f_{2p}$.

The morphisms $g_{1e} \circ e_1 : PAR_1 \rightarrow EXP_3$ and $g_{2e} \circ e_2 : PAR_2 \rightarrow EXP_3$ satisfy the following

$$g_{1e} \circ e_1 \circ f_{1p} = g_{1e} \circ f_{1e} \circ e_0 \quad (1)$$
$$= g_{2e} \circ f_{2e} \circ e_0 \quad (2)$$
$$= g_{2e} \circ e_2 \circ f_{2p} \quad (3)$$

(1) is obtained by assumption on $f_{1e}$ and $e_0$

(2) is obtained by pushout property of $EXP_3$

(3) is obtained by assumption on $f_{2e}$ and $f_{2p}$

Hence, by the universal pushout property of $PAR_3$, there exist a unique morphism $e_3 : PAR_3 \rightarrow EXP_3$ such that $g_{1e} \circ e_1 = e_3 \circ g_{1p}$ and $g_{2e} \circ e_2 = e_3 \circ g_{2p}$.

A similar reasoning defines $i_3, s_3, v_3$. The constraint $v_3 \circ e_3 = s_3 \circ i_3$ is a consequence of the uniqueness of the specification morphism $PAR_3 \rightarrow BOD_3$ induced by the universal pushout property of $PAR_3$. Therefore, the diagram 6.8 commutes.

### 6.3.4 Actualization of Module

Actualization of a module replaces its parameter part by an actual parameter which is given by a specification or a parameterized specification.

**Definition 6.5. Actualization.**

Given a module specifications $MOD = \langle PAR, EXP, IMP, BOD, e, s, i, v \rangle$, a parameterized specification $PSPEC_1 = \langle PAR_1, ACT_1 \rangle$ and a specification morphism $h : PAR \rightarrow ACT_1$, the actualization $MOD_1 = \langle PAR_1, EXP_1, IMP_1, BOD_1, e_1, s_1, i_1, v_1 \rangle$ of $MOD$ by $PSPEC_1$ and $h$ is defined as follows:

- $EXP_1 = EXP +_{PAR} ACT_1$
- $IMP_1 = IMP +_{PAR} ACT_1$
- $BOD_1 = BOD +_{EXP} EXP_1$

The specification morphisms $e_0, s_1, i_0$ and $v_1$ are uniquely defined by the pushout properties in the following diagram 6.9, while $e_1 = e_0 \circ j$ and $i_1 = i_0 \circ j$ where $j : PAR_1 \rightarrow ACT_1$ is the inclusion of $PAR_1$ into $ACT_1$. 

We note \( MOD_1 = MOD_B(PSPEC_1) \)

The actualized module specification \( MOD_1 \) receives its parameter \( PAR_1 \) from the parameterized \( PSPEC_1 = \langle PAR_1, ACT_1 \rangle \). If \( PAR_1 \) is the empty specification, \( MOD_1 \) is a non-parameterized module specification. In both case, \( EXP, IMP, \) and \( BOD \) of \( MOD \) are actualized by \( ACT_1 \) via \( h \). This means that \( EXP_1, IMP_1, \) and \( BOD_1 \) are obtained from \( EXP, IMP, \) and \( BOD \) by replacing the formal parameter \( PAR \) by the actual parameter \( ACT_1 \).

### 6.3.5 Extension of Module

The extension \( ext_E(MOD) \) of a module \( MOD \) is the result of extending some or all constituent parts of a module by additional items, where \( E \) denotes the collection of all extended items. This extension must not change the given module \( MOD \) which is assumed to become a submodule of \( ext_E(MOD) \). This construction is very important to build up modules step by step. In general, the extended part \( E \) itself, is not a module or a parameterized data type. This means that only the extended module \( ext_E(MOD) \) but not the extended part \( E \), will be a component of the modular system.

**Definition 6.6. Extension.**

Given a module specifications \( MOD = \langle PAR, EXP, IMP, BOD, e, s, i, v \rangle \), where all its specification morphisms are inclusions, and a four-tuple \( E = \langle E_{PAR}, E_{EXP}, E_{IMP}, E_{BOD} \rangle \) of extended items, the extension \( MOD_1 = \langle PAR_1, EXP_1, IMP_1, BOD_1, e_1, s_1, i_1, v_1 \rangle \) of \( MOD \) by \( E \), denoted by \( ext_E(MOD) \), is a module specification containing the module \( MOD \) as a submodule specification. More precisely, we have:

- \( PAR_1 = PAR + E_{PAR} \)
- \( EXP_1 = EXP + E_{\text{PAR}} + E_{\text{EXP}} \)
- \( IMP_1 = IMP + E_{\text{PAR}} + E_{\text{IMP}} \)
- \( BOD_1 = BOD + E_{\text{PAR}} + E_{\text{EXP}} + E_{\text{IMP}} + E_{\text{BOD}} \)
- The specification morphisms \( e_1, s_1, i_1 \) and \( v_1 \) are such that the diagram 6.10 commutes.

![Diagram](attachment:image.png)

**Figure 6.10: Extension of a Module Specification**

In this section, we present the four interconnection kinds and their associated operators, defined by Ehrig and Mahr [25].

### 6.4 Weaving Operator

This section is intended to define the weaving process by using categorical concepts and operators defined in section 6.3. Weaving is the predominant notion of model combination in Aspect Oriented Software Development (AOSD) [34]. Weaving uses a binary strategy because a weaver weaves an aspect into the base component at a time. Binary strategies allow merging two models at a time. To merge more than two models, the result of merging two models is considered as a new model and then merged with another model. The weaving process is a kind of the model merging. Model merging is useful in any conceptual modeling language as a way of consolidating a set of models to gain a unified perspective, to understand interactions among models, or to perform various types of end-to-end analysis [89]. In our formalism, models are aspect-class module specifications and aspect-class relationships are structure-preserving mappings, i.e. aspect-class morphisms. The mappings express ways in which the structure of one module maps onto that of another. Model merging has some requirement criteria which are [89]:
• **Non-redundancy:** Only one copy of the common parts is included in the merged model.

• **Completeness:** Merge should not lose information, i.e., it should represent all the source models completely.

• **Minimality:** Merge should not introduce information that is not present in or implied by the source models.

• **Semantic Preservation:** Merge should support the expression and preservation of semantic properties. For example, if models are expressed as state machines, one may want to preserve their temporal behaviors to ensure that the merge properly captures the intended meaning of the source models.

The weaving operator we describe in this section is based on category concepts and operators described in section 6.3, which use the notion of colimit. The colimit of an interconnection diagram is a new object, called the colimiting object, together with a family of mappings, one from each object in the diagram onto the colimiting object. Since each mapping expresses how the internal structure of its source object is mapped onto that of its destination object, the colimit expresses the merge of all the objects in the interconnection diagram. Furthermore, the colimit respects the mappings in the diagram: The intuition here is that the image of each object in the colimit is the same, no matter which path through the mappings in the diagram is followed. By definition, the colimit is also minimal; it merges the objects in the diagram without adding anything essentially new. Hence, the use of the notion of colimit is an advantage of our weaving formalism because of the merging requirements quoted above, which are present in the notion of colimit, are also required in the weaving process in AOSD.

We have identified three ways to define the weaving operator:

1. We can consider that aspect and class components which are related by a morphism share some elements, joinpoints. In other words, the base component and an aspect overlap over a set of joinpoints contained in the pointcuts of the aspect. Thus, we can use the union operation to write the weaving algorithm.

2. A class can be seen as a client and an aspect as a server which provides services to this client [72]. Then, we can use the composition operation to model the weaving process.
3. We can view aspect as a program transformation [54]. A program transformation is a function that maps programs to programs (in [54]). We can thus use actualization operation to formalize the weaving process.

We use the first idea to develop our weaving algorithm, i.e., the weaving algorithm is based on the three-way merging technique. We will explore the two other ideas in future works.

6.4.1 Three Way Merging

Three-way merging attempts to merge two models by relying on the common parts of these two models, i.e., the information in the common part is used during the merge process [57]. If we want to merge two models, A and B, that overlap in some way, we can express the overlap as a third model, C, with mappings from C to each of A and B:

\[
\begin{array}{c}
A \\
\downarrow f \\
C \\
\uparrow g \\
B
\end{array}
\]

In this interconnection diagram, the two mappings \( f \) and \( g \) specify how the common part, \( C \), is represented in each of \( A \) and \( B \). The colimit of this diagram is a new model, \( P \), expressing the union of \( A \) and \( B \), such that their overlap, \( C \), is included only once. We call the model \( C \) a connector model. Our weaving formalism is based on the three-way merging technique.

6.4.2 Required Definitions

We give in this section the definitions of required concepts used in our weaving algorithm.

6.4.2.1 Ordered Module Specification

The following definitions use the notion of "order" 4.3 defined in chapter 4.

**Definition 6.7.** *Ordered signature.*

Let \( \Sigma \) be an algebraic signature. \( \Sigma \) is an ordered signature iff there is an order \( \leq_o \) defined over \( \Sigma \).
**Definition 6.8.** Ordered specification.
An algebraic specification $SPEC$ is ordered iff its signature is an ordered signature with
an order $\leq_o$, and the set of axioms is ordered by $\leq_o$.

**Definition 6.9.** Ordered module specification.
A module specification is ordered iff its four algebraic specifications are ordered.

**Remark 6.4.**
- At the implementation level, an order $\leq_o$ corresponds to the elements execution order in a program execution flow.
- $a =_e b$ means that $a$ and $b$ appears simultaneously.
- In the sequel, we assume that all module specifications of a system are ordered.

**Theorem 6.1.** The ADT, Class and Aspect module specifications are ordered.

**Proof.** We can define an order $\leq_o$ on a module specification $M$ as follows:
$a \leq_o b$ iff in $M$, the line of $a$ is immediately before the line of $b$.

### 6.4.2.2 Multiset

A multiset is a generalization of a set. While each element of a set has only one instance, an element of a multiset can have more than one instance. The total number of elements in a multiset, including repeated memberships, is the cardinality of the multiset. The number of times an element belongs to the multiset is the multiplicity of that member. For example, in the multiset $\{a, a, b, b, b, c\}$ the multiplicities of the members $a, b,$ and $c$ are respectively 2, 3, and 1, and the cardinality of the multiset is 6. To distinguish between sets and multisets, a notation that incorporates brackets is sometimes used: the multiset $\{2, 2, 3\}$ can be represented as $[2, 2, 3]$. We need this multiset concept because a base program can expose a same joinpoint many times (e.g., a method can be called more than one time in a program). Formally, a multiset can be defined as a pair $\langle A, m \rangle$ where $A$ is some set and $m$ is a function from $A$ to the set $\mathbb{N}$ of positive natural numbers. The set $A$ is called the underlying set of elements. For each $a$ in $A$ the multiplicity (that is, number of occurrences) of $a$ is the number $m(a)$.

### 6.4.3 Weaving Algorithm

Suppose we want to weave a class component and an aspect component. Let the class component represented by the module specification
\[ MOD_c = \langle PAR_c, EXP_c, IMP_c, BOD_c \rangle \] and the aspect component by \[ MOD_a = \langle PAR_a, EXP_a, IMP_a, BOD_a \rangle \]. We first define a connector module specification which contains the common part of the aspect and the class module specifications. Then, we specify through morphisms how this connector is embedded in the source modules. Afterwards, we apply the weaving algorithm to compute the augmented module. Let \( MOD_s = \langle PAR_s, EXP_s, IMP_s, BOD_s \rangle \) be the connector (shared) module specification, \( f : MOD_s \rightarrow MOD_a \) and \( g : MOD_s \rightarrow MOD_c \) the morphisms between the connector module and the aspect and class modules respectively.

\[
\begin{array}{c}
MOD_a \\
g \downarrow \\
MOD_s \\
f \downarrow \\
MOD_c
\end{array}
\]

Denoting the weaving operator as \( w \), we write \( MOD_w = MOD_a w MOD_c \) and we read \( MOD_w \) is the woven module of \( MOD_a \) and \( MOD_c \). Let \( MOD_w = \langle PAR_w, EXP_w, IMP_w, BOD_w \rangle \) be the augmented module specification. We note an advice of an aspect as \( Ad = \langle \text{mod, TJ, } \alpha \rangle \) where:

- \( \text{mod} \) is the modifier of the advice \( Ad \). \( \text{mod} \) belongs to the set \( \text{MODIFIER} = \{ \text{before, after, around, insteadOf} \} \)

- \( TJ \) is a multiset of the joinpoints containing in the pointcut of the advice \( Ad \);

- \( \alpha \) is an action representing the body of the advice;

\( Ad.mod \) gives the modifier \( mod \) of the advice, \( Ad.TJ \) gives \( TJ \), and \( Ad.\alpha \) gives \( \alpha \). We note TAD the set of all advices of an aspect. For an aspect \( A \), \( A.\text{TAD} \) gives this set TAD. The weaving algorithm starts by computing the union (using union operator 6.4) of the module specifications \( MOD_a \) and \( MOD_c \) (line 2). If the aspect does not contain any advice, \( MOD_a \) is empty and the augmented module \( MOD_w \) is equal to \( MOD_a \) (line 4). For each advice in the aspect module, we consider the four kinds of the advice. For example, if the advice type is \( \text{before} \), for each joinpoint \( jp \) related to this advice, if \( jp \) is selected (line 9) and the apparition of \( jp \) is before the apparition of the advice body in \( M_a \), we permute this joinpoint and this advice body (line 10) s.t. the advice body is the immediate predecessor of the joinpoint. The algorithm does the same thing for the \( \text{after} \) and around advices. For around advice, \( \alpha_1 \) is the action inserted before \( jp \)
1: Begin weaving
2: \( M_u \leftarrow MOD_a + (MOD_{a,f,g}) MOD_c \)
3: if \( MOD_a.TAD = \emptyset \) then
4: \( MOD_w \leftarrow M_u \)
5: end if
6: for all \( Ad \) in \( MOD_a.TAD \) do
7: if \( Ad.mod = \text{before} \) then
8: for all \( jp \) in \( Ad.TJ \) do
9: if \( jp \in MOD_s \) then
10: permut \( jp \) and \( Ad.\alpha \) s.t. \( Ad.\alpha <^i_o f_j(jp) \)
11: end if
12: end for
13: else if \( Ad.mod = \text{after} \) then
14: for all \( jp \) in \( Ad.TJ \) do
15: if \( jp \in MOD_s \) then
16: permut \( jp \) and \( Ad.\alpha \) s.t. \( f_j(jp) <^i_o Ad.\alpha \)
17: end if
18: end for
19: else if \( Ad.mod = \text{around} \) then
20: for all \( jp \) in \( Ad.TJ \) do
21: if \( jp \in MOD_s \) then
22: permut \( jp \) and \( Ad.\alpha = \alpha_1\alpha_2 \) s.t.
23: \( \alpha_1 <^i_o f_j(jp) <^i_o \alpha_2 \)
24: end if
25: end for
26: else
27: \( M_u \leftarrow \text{replaceAll}(M_u, Ad.TJ, Ad.\alpha) \)
28: end if
29: end for
30: remove from \( M_u \), pointcut and advice elements, except selected joinpoints and advice bodies related to base module.
31: replace in \( M_u \) selected joinpoints \( jp \) by their associated actions \( f_j(jp) \).
32: \( MOD_w \leftarrow M_u \)
33: End weaving

Algorithm 1: Weaving Algorithm
and $\alpha_2$ the action after $jp$. For the insteadOf advice, we replace all occurrences of the selected joinpoints by the advice body (line 26). At line 29, we remove all pointcut and advice elements (e.g., pointcut names, advice names) except selected joinpoints and advice bodies. At line 30, we replace the selected joinpoints by their associated actions by using the function $f_j: J \longrightarrow \mathcal{A}$ that associates to a joinpoint $jp$, the action $f_j(jp)$ encapsulated in the body of $jp$; $J$ is a set of joinpoints and $\mathcal{A}$ a set of actions.

Let $T$ be a multiset containing all (matching) joinpoints between an aspect module and a class module. $M' = replaceAll(M, T, \alpha)$ means that we get the module $M'$ by replacing in the module $M$ all occurrences of each element of $T$ by the action $\alpha$.

```plaintext
1: Begin replaceAll($M, T, \alpha$)
   \{ $M = \langle PAR, EXP, IMP, BOD \rangle$ \}
2: for all $e$ in $T$ do
3:   if $e$ is in $PAR \cup EXP \cup IMP \cup BOD$ then
4:     replace $e$ by $\alpha$ in $M$
5:   end if
6: end for
7: End replaceAll
```

**Algorithm 2**: Function replaceAll

Our algorithm works because union operation works. Many research proposals have explored different mechanisms for specifying and weaving aspects into UML design models (for instance one in [65]). Other used UML models and graph transformation. The advantage of our weaving approach over these approaches is the use of colimit concepts, with the above properties, and which allow a formal modular reasoning.

**Remark 6.5.** Since the weaving operator is mainly based on the union operator, its commutative diagram is the commutative diagram of the union operator in which certain arrangement should be made.

### 6.4.4 The Definition of the Connector Module

A connector module specification $MOD_s = \langle PAR_s, EXP_s, IMP_s, BOD_s \rangle$ does not provide any service; it just a link between an aspect module $MOD_a = \langle PAR_a, EXP_a, IMP_a, BOD_a \rangle$ and a class module $MOD_c = \langle PAR_c, EXP_c, IMP_c, BOD_c \rangle$. $MOD_s$ is a shared submodule of $MOD_a$ and $MOD_c$. We define $BOD_s$ as a part containing only auxiliary elements. These auxiliary elements are joinpoints body contained in $MOD_a$, which are included in $MOD_c$. 
\[ \text{PAR}_s = \text{PAR}_a \cap \text{PAR}_c \]
\[ \text{EXP}_s = \text{EXP}_a \cap \text{EXP}_c \]
\[ \text{IMP}_s = \text{IMP}_a \cap \text{IMP}_c \]

\[ \text{BOD}_s = (\text{BOD}_a \cap \text{BOD}_c \cap \{jp \in \text{Ad.TJ}, \forall \text{Ad} \in \text{MOD}_a.\text{TAD}) \} \cup \{\text{shared sorts}\}. \]

The four specification morphisms are inclusion morphisms.

The two module specification morphisms \( f : \text{MOD}_s \rightarrow \text{MOD}_a \) and \( g : \text{MOD}_s \rightarrow \text{MOD}_c \) are inclusion morphisms, i.e., \( f(x) = x \) and \( g(x) = x \). For clarity purpose, we add a field \textbf{Joinpoints} to the pattern of an aspect specification. This field will contain all joinpoints of the aspect component.

\textit{Remark 6.6.} It is worth to notice that a joinpoint \( x \) of \( \text{MOD}_s \) can be related to an element \( y \) of \( \text{MOD}_c \) by the morphism \( g \), no matter of the field of \( \text{MOD}_c \) \( y \) belongs to.

6.4.5 Example

Consider a classical example for figure editor system of the literature defined in chapter 2, section 3.1. Consider the example of the Class Point of the figure 3.2. Its module specification is \( \text{MOD}_p = \langle \text{PAR}_p, \text{EXP}_p, \text{IMP}_p, \text{BOD}_p \rangle \) with the four inclusion morphisms \( e_p, s_p, i_p, v_p \) such that (figures 6.11, 6.12, 6.13, 6.14):

<table>
<thead>
<tr>
<th>SPEC_CLASS</th>
<th>PAR_p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>integer</td>
</tr>
<tr>
<td>END PAR_p</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPEC_CLASS</th>
<th>IMP_p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>integer</td>
</tr>
<tr>
<td>Actions</td>
<td>(classical operations on integer)</td>
</tr>
<tr>
<td>END IMP_p</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.11: Module specification of Class Point, Part1
**SPEC_CLASS** \(EXP_p\)

**Sorts:** integer, point

**States:**
- \(x: \text{integer} \rightarrow \text{integer}\)
- \(y: \text{integer} \rightarrow \text{integer}\)

**Actions:**
- set\(X\): integer
- set\(Y\): integer

**END** \(EXP_p\)

Figure 6.12: Module specification of Class Point, Part 2

**SPEC_CLASS** \(BOD_p\)

**Sorts:** integer, point

**States:**
- \(x: \text{integer} \rightarrow \text{integer}\)
- \(y: \text{integer} \rightarrow \text{integer}\)

**Actions:**
- set\(X\): integer
- set\(Y\): integer
  (classical operations on integer)

**Axioms:**
- \(G(\forall t, [\text{set\(X\)}(t)][x = t])\)
- \(G(\forall t, [\text{set\(Y\)}(t)][y = t])\)

**End** \(BOD_p\)

Figure 6.13: Module specification of Class Point, Part 3

\[\begin{align*}
  e_p & : PAR_p \rightarrow EXP_p \\
  i_p & : PAR_p \rightarrow IMP_p \\
  v_p & : EXP_p \rightarrow BOD_p \\
  s_p & : IMP_p \rightarrow BOD_p
\end{align*}\]

\textbf{where} \(v_p \circ e_p = s_p \circ i_p\) is satisfied.

Figure 6.14: Module specification of Class Point, Part 4

The module specification \(MOD_D = \langle PAR_D, EXP_D, IMP_D, BOD_D \rangle\) with the four
inclusion morphisms \( e_d, s_d, i_d, v_d \) of the aspect \( \text{DisplayUpdating} \) is defined in the figures 6.15, 6.16, 6.17, 6.18 and 6.19.

**Figure 6.15:** Module specification of Aspect \( \text{DisplayUpdating} \), Part 1.

```
SPEC_ASPECT \( PAR_D \)
  Sorts: integer
  Actions:
  update
END \( PAR_D \)
```

**Figure 6.16:** Module specification of Aspect \( \text{DisplayUpdating} \), Part 2.

```
SPEC_ASPECT \( IMP_D \)
  Sorts: integer, point, line, display
  Joinpoints: p: point, l: line
    execution(p.setX()), execution(p.setY()),
    execution(l.setP1()), execution(l.setP2())
  Actions:
    update
    (classical operations on integer)
END \( IMP_D \)
```

**Figure 6.17:** Module specification of Aspect \( \text{DisplayUpdating} \), Part 3.

```
SPEC_ASPECT \( EXP_D \)
  Sorts: integer, displayUpdating
  Pointcuts:
    move: integer point \( \rightarrow \) bool
  Actions:
    update
END \( EXP_D \)
```
SPEC_ASPECT \( BOD_D \)

**Sorts:** displayUpdating, point, line, display, integer

**Joinpoints:** p: point, l: line
- execution(p.setX()), execution(p.setY()),
- execution(l.setP1()), execution(l.setP2())

**Pointcuts**
- move

**Actions:**
- update

**Pointcut Axioms:** p, p’: point, l: line, t: integer
\[
\forall t, p, p’, l, t,
\text{move}: \text{pr}(\text{execution}(p.setX(t))) \lor
\text{pr}(\text{execution}(p.setY(t))) \lor
\text{pr}(\text{execution}(l.setP1(p’))) \lor
\text{pr}(\text{execution}(l.setP2(p’)))
\]

**Advice Axioms:** p, p’: point, l: line,
- t: integer, d: display
\[
\forall t, p, p’, l, t,
(A_f \text{pr}(\text{execution}(p.setX(t)))) \implies \text{d.update}) \lor
(A_f \text{pr}(\text{execution}(p.setY(t)))) \implies \text{d.update}) \lor
(A_f \text{pr}(\text{execution}(l.setP1(p’)))) \implies \text{d.update}) \lor
(A_f \text{pr}(\text{execution}(l.setP2(p’)))) \implies \text{d.update})
\]

End \( BOD_D \)

Figure 6.18: Module specification of Aspect DisplayUpdating, Part 4.

\[
e_d : PAR_D \rightarrow EXP_D
\]
\[
i_d : PAR_D \rightarrow IMP_D
\]
\[
v_d : EXP_D \rightarrow BOD_D
\]
\[
s_d : IMP_D \rightarrow BOD_D
\]

where \( v_d \circ e_d = s_d \circ i_d \) is satisfied.

Figure 6.19: Module specification of Aspect DisplayUpdating, Part 5.

Let \( MOD_S = \langle PAR_S, EXP_S, IMP_S, BOD_S \rangle \) be the connector module of \( MOD_P \) and \( MOD_D \) with the four inclusion morphisms \( e_s, s_s, i_s, v_s \) such that:

\[
PAR_S = PAR_D \cap PAR_P
\]
\[
EXP_S = EXP_D \cap EXP_P
\]
\[
\text{IMP}_S = \text{IMP}_D \cap \text{IMP}_P
\]

\[
\text{BOD}_S = (\text{BOD}_D \cap \text{BOD}_P \cap \{jp \in \text{Ad.TJ}, \forall \text{Ad} \in \text{MOD}_D.TAD\}) \cup \{\text{shared sorts}\}.
\]

\(\text{MOD}_{S_1}\) is represented in the figures 6.20, 6.21 and 6.22.

```
SPEC_CLASS PAR_S
  Sorts: integer
END PAR_S

SPEC_CLASS EXP_S
  Sorts: integer
END EXP_P
```

Figure 6.20: Module specification of Class Point: Part 1

```
SPEC_CLASS IMP_S
  Sorts: integer
  Actions:
    (classical operations on integer)
END IMP_S
```

Figure 6.21: Module specification of Class Point: Part 2

```
SPEC_CLASS BOD_S
  Sorts: integer, point
  Joinpoints: p: point
    execution(p.setX()),
    execution(p.setY())
End BOD_S

\(e_s : \text{PAR}_S \rightarrow \text{EXP}_S\)
\(i_s : \text{PAR}_S \rightarrow \text{IMP}_S\)
\(v_s : \text{EXP}_S \rightarrow \text{BOD}_S\)
\(s_s : \text{IMP}_S \rightarrow \text{BOD}_S\)

\text{where} \ v_s \circ e_s = s_s \circ i_s \text{ is satisfied.}
```

Figure 6.22: Module specification of Class Point: Part 3
The two module specification morphisms \( f : MOD_S \rightarrow MOD_D \) and \( g : MOD_S \rightarrow MOD_P \) are inclusion morphisms.

By applying the algorithm 1, we first compute \( M_u \) and then \( MOD_w \). \( M_u = \langle PAR_u, EXP_u, IMP_u, BOD_u \rangle \) is defined by the figures 6.23, 6.24 and 6.25.

```
SPEC_ASPECT PAR_u
  Sorts: integer
  Actions:
    update
END PAR_u

SPEC_ASPECT IMP_u
  Sorts: integer, point, line, display
  Joinpoints: p: point, l: line
    execution(p.setX()), execution(p.setY()), execution(l.setP1()), execution(l.setP2())
  Actions:
    update
    (classical operations on integer)
END IMP_u

SPEC_ASPECT EXP_u
  Sorts: integer, displayUpdating, point
  Pointcuts:
    move: integer point \rightarrow bool
  Actions:
    setX: integer
    setY: integer
    update
END EXP_u
```

Figure 6.23: Module specification \( M_u \), Part 1.
SPEC_ASPECT  $BOD_u$

Sorts: display Updating, point, line, display, integer

States:
  x: integer $\to$ integer
  y: integer $\to$ integer

Joinpoints: l: line
  $\forall l$ execution(l.setPl()), execution(l.setP2())

Pointcuts
  move

Actions:
  setX: integer, setY: integer, update
  (classical operations on integer)

Pointcut Axioms:
  p, p': point, l: line, t: integer
  $\forall t, p, p', l, t,$
  move: pr(execution(p.setX(t)))V
  pr(execution(p.setY(t)))V
  pr(execution(l.setPl(p')))V
  pr(execution(l.setP2(p')))V

Advice Axioms:
  p, p': point, l: line, t: integer, d: display
  $\forall t, p, p', l, t,$
  (Af pr(execution(p.setX(t))) d.update)V
  (Af pr(execution(p.setY(t))) d.update)V
  (Af pr(execution(l.setPl(p'))) d.update)V
  (Af pr(execution(l.setP2(p'))) d.update)

Axioms:
  t: integer
  $G(\forall t, \text{[setX(t)]}(x = t)), \ G(\forall t, \text{[setY(t)]}(y = t))$

End $BOD_u$

Figure 6.24: Module specification $M_u$, Part 2.

\begin{align*}
e_u : PAR_u & \to EXP_u \\
l_u : PAR_u & \to IMP_u \\
v_u : EXP_u & \to BOD_u \\
s_u : IMP_u & \to BOD_u
\end{align*}

\textit{where } $v_u \circ e_u = s_u \circ l_u \text{ is satisfied.}$

Figure 6.25: Module specification $M_u$, Part 3.
The aspect contains only one advice and its modifier is equal to after. Thus, the lines of the algorithm 1 which are concerned are line 13 to 18. Then, we apply lines 29 and 31 to get the augmented module $MOD_w = \langle PAR_w, EXP_w, IMP_w, BOD_w \rangle$. $MOD_w$ is defined by the figures 6.26, 6.27, 6.28, 6.29 and 6.30.

```
SPEC_CLASS PAR_w
  Sorts: integer
  Actions:
    update
END PAR_w
```

Figure 6.26: Module specification $MOD_w$, Part 1.

```
SPEC_CLASS IMP_w
  Sorts: integer, point, line, display
  Actions:
    update
      (classical operations on integer)
END IMP_w
```

Figure 6.27: Module specification $MOD_w$, Part 2.

```
SPEC_CLASS EXP_w
  Sorts: integer, displayUpdaring, point
  Actions:
    setX: integer
    setY: integer
    update
END EXP_w
```

Figure 6.28: Module specification $MOD_w$, Part 3.
\textbf{SPEC\_CLASS} \hspace{0.5cm} BOD_w

\textbf{Sorts:} displayUpdating, point, line, display, integer

\textbf{States:}
\begin{itemize}
  \item x: integer $\rightarrow$ integer
  \item y: integer $\rightarrow$ integer
\end{itemize}

\textbf{Actions:}
\begin{itemize}
  \item setX: integer
  \item setY: integer
\end{itemize}

(classical operations on integer)

update

\textbf{Axioms:}
\begin{itemize}
  \item t: integer
  \item $G(\forall t, [setX(t)](x = t))$
  \item $G(\forall t, [setY(t)](y = t))$
\end{itemize}

End BOD_w

Figure 6.29: Module specification \textit{MOD}_w, Part 4.

\begin{align*}
e_w : \text{PAR}_w & \rightarrow \text{EXP}_w \\
i_w : \text{PAR}_w & \rightarrow \text{IMP}_w \\
v_w : \text{EXP}_w & \rightarrow \text{BOD}_w \\
s_w : \text{IMP}_w & \rightarrow \text{BOD}_w
\end{align*}

where $v_w \circ e_w = s_w \circ i_w$ is satisfied.

Figure 6.30: Module specification \textit{MOD}_w, Part 5.

6.5 Conclusion

In this chapter, we described the societal life of software components belonging to the software architecture of a computing system. We described the existing interconnection mechanisms and their associated operations. We defined a new interconnection relation, the weaving relation, and its associated operation. This operation is defined by the algorithm 1. This operation uses the notion of colimit, the union operation. The use of colimit concepts is an advantage of our weaving algorithm, because the properties of colimit are what are required by a weaving process in AOSD. To the best of our knowledge, there are very few aspect-oriented approaches that formally define the weaving process.
Besides, we identify two other interesting ideas which can be used to define the weaving operation. The first one is inspired from the paper [72]. A class module specification can be seen as a client and an aspect module as a server which provides services to this client. Therefore, we can assimilate an aspect-class relationship to a client/server (use) relationship defined in the section 6.3 and then compute the augmented module specification by using the composition operation. The second idea comes from the paper [54]. We can view aspect as a model (or program) transformation. An aspect is a declaration of changes that are to be made to a model or program. Weaving is the process that makes these changes. A program transformation is a function that maps programs to programs (in [54]). We can thus use actualization operation to formalize the weaving process. But, we have to redefine this actualization operation in such way we can actualize a generic module by a module, instead of an actualization of a module by an algebraic specification defined in chapter 6.3. We let the exploration of these two ideas for the future works.
CHAPTER 7

ASPECT PREVENTION MECHANISM

7.1 Introduction

Aspect interaction is one of the main concerns in the aspect-oriented community [23]. The aspect interaction problem can be presented as follows [96]:

- Let $P_1$ be a property satisfied by aspect $A$.
- Let $P_2$ be a property satisfied by the composition of the base system and $n$ aspects (SYS), where $n$ is a positive integer. $A$ is not yet composed with SYS.
- Both $P_1$ and $P_2$ should remain satisfied by the composition of $A$ and SYS; if not, then, there is an interaction problem.

The interaction between aspects and base classes may introduce a variety of bug into the system. Detection and resolution of undesirable aspect interactions is an important open research field. Most aspect-oriented verification approaches are based on a detection and correction strategy. Although these detection approaches are relevant for aspect-oriented software reliability, we believe that they are time and cost consuming. It is good to detect and correct system failures, but it is better to prevent these failures. Contrary to the existing approaches, we adopt a prevention policy in our approach. We integrate this prevention policy at the specification phase. We believe indeed that this will make the verification phase must faster and cheaper. We can convince ourselves by making an analogy to the medicine, where experts and governments prefer to place more emphasis on measures to prevent disease. The same analogy could be observed in avionics where MRO (maintenance, repair, overhaul) activities put more emphasis on preventive actions than correctives actions for flight quality and reliable aircraft. With a prevention policy, the number of properties that must be verified on an aspect-oriented system will be smaller than the case in which this prevention policy has not be taken into account.

An initial aspect faults model developed in [5] lists the main fault types that can arise in aspect-oriented applications due to the aspect integration into the base systems. This initial model has been refined in [7, 27, 53]. Undesirable aspect interactions are characterized, among others \(^1\), by these fault types. For instance, one of the fault types

\(^1\)since other sources of error may exist.
is: "Incorrect aspect precedence" which can arise when more than one aspect affects
the same joinpoint (a joinpoint is a place in a base component where an aspect can be
applied). Existing verification approaches can detect only one or two of these fault types.
To the best of our knowledge, there are no existing methods or tools capable of taking all
of them into account. An aspect verification approach survey [46] also points out this.
There is a lack of efficient mechanisms for dealing with aspect interactions, and which
can take most of these aspect fault types into account.

The logic $L_A$ that we presented in chapter 4 will help us specify system components
with a prevention mechanism, which will prevent most (as many as possible) of the
undesirable aspect interactions characterized by these fault types. In our approach, we
adopt the preventive and corrective strategies to deal with aspect interaction problems.
A preventive strategy is based on preventing a non-conformant event in the future. A
corrective strategy is based on a non-conformant event that has happened in the past.
Our prevention mechanism plays the role of the prevention action. Corrective action will
be taken at the verification phase when it is necessary, i.e., after an interaction problem
has occurred.

The rest of this chapter is organized as follows. Section 7.2 presents the AOP fault
model on which our prevention policy is based. Section 7.3 presents our prevention
mechanism. Our verification strategy is presented in the section 7.4. Section 7.5 con-
cludes this chapter.

7.2 A fault model for AOPs

Alexander and Bieman [5] propose an initial fault model for AOP. This fault model
is based on the nature of faults and the failures in AOP. The pointcut fault type of this
model contains two cases. Authors of the paper [53] added two more cases at this fault
type. Then, authors of the papers [7, 27] defined advice related fault types. However,
in our thesis, we consider the initial model of Alexander and Bieman, with the two more
cases defined in [53]. In the future work, we will deal with the advice fault type and
other that will be emerged in future. In this section, we present the fault model on which
our prevention policy is based.
7.2.1 Incorrect Aspect Precedence

In an Aspect Oriented Program, it is possible for more than one advice to be affecting the same join point. Aspects are typically constructed by pointcuts and advice. A pointcut selects a set of join points. Join point defines certain interaction points ("hooks") where the original behavior of the program can be modified or enhanced, typically by superimposing, or weaving, additional behavior [63]. Advice consists of the units of execution that are inserted at these join points (see section 3.1 of the chapter 2 for details). When multiple aspects are composed undesired behavior may emerge due to the interference of aspects. This phenomenon is known as one type of aspect interference problem (in general, an interference problem happens when interaction changes an aspect’s behavior or disables an aspect). There are different causes of this kind of aspect’s interference:

- At weave-time, aspects that change the static structure of a program (introductions) can cause ambiguous weaving - resulting in different programs - depending on the weaving order [4].

- At run-time, one aspect can change or abort the control-flow of the system, causing a join point of another aspect to never be reached [4].

- Aspect interference occurs if one aspect writes to variables or fields, such that the behavior of a succeeding advice is affected. As this is a general problem, it can also occur on shared join points [4].

- Semantically, composing an aspect Aspect₂ before an aspect Aspect₁ at the same join point (instead of Aspect₁ before Aspect₂) could lead to nonsense depending on the application domain.

When no fixed order of advice execution is determined by the program directives, but the order of advice execution affects the result, a shared join point can lead to unpredictable and undesired behavior of the woven system, or even an ambiguous system. Join points that are selected by more than one pointcut are called shared join points. Other studies have already indicated that special attention must be paid to shared join points (in [4]).

7.2.2 Incorrect Strength in Pointcut Patterns

The syntactic definition of an aspect consists of a set of patterns, referred to as pointcuts, that are used to select elements, called join points, that appear in the bodies of
methods. These join points correspond to well defined locations in the control flow of a method of some concern. Pointcuts contain specifications that select join points of a particular type according to a signature that includes a pattern. For example, a pointcut pointcut creditOps1(): call(*Account.credit()); might capture the calls to all methods of an Account class named credit. In this example, call is the type of joinpoints, and *Account.credit() is the signature pattern of the joinpoints (arguments are included in the signature). Now, consider another example of pointcut pointcut creditOps2(): call(*Account+.credit(..));. The pointcut creditOps2() captures the calls of all methods named credit, of the Account class and all its sub-types. The pattern in the pointcut creditOps1() is said to be stronger than the pattern in second pattern creditOps2(), because the first one is more restrictive than the second one. If the pattern is too strong, some necessary joinpoints will not be selected (i.e., select a smaller set than the adequate set of joinpoints). If the pattern is too weak, additional join points will be selected that should be ignored (i.e., select a bigger set than the adequate set of joinpoints). These two cases of the pointcut fault type are described in figure 7.1 C and D. Cases C and D along with two other cases A and B (figure 7.1) of this fault type have been identified in [53]. (A): select a set of joinpoints which has some intersection with the adequate set but also selects other joinpoints (i.e selects some of the intended joinpoints but also some unintended joinpoints); (B) select a different set, with no intersection with the adequate set of joinpoints (i.e selects none of the intended joinpoints). That is a major type of aspect interaction problem.

7.2.3 Failure to Establish Expected Post-Conditions

Aspects can cause changes in the flow of control of a class’s code (a class of the base program). Such a change in flow of control can result in a class (core concern) not being able to fulfill the post-conditions of its class contract. The clients of core concerns expect those concerns to behave according to their contracts. A client has the responsibility to ensure that a method precondition holds prior to calling the method. Given that the precondition is ensured, the client can reasonably assume that the method’s post-conditions will be satisfied (total correctness). Clients expect method post-conditions to be satisfied regardless of whether or not aspects are woven into the concern. Hence the behavioral contracts of the concern should hold after the weaving process. Thus, for correct behavior, woven advice must allow methods in core concerns to satisfy their post-conditions. Defining advice that does not cause behavioral contracts to be broken
in all likely weave contexts and with all likely combinations of aspects will be a difficult challenge for aspect developers, and a likely source of errors.

7.2.4 Failure to Preserve State Invariants

A concern’s behavior is defined in terms of a physical representation of its state, and methods that act on that state. The integration of an aspect into the base program can introduce new methods and instance variables into the core concerns (classes). Thus, this integration can introduce new states and can cause the classes to violate their state invariants. In addition to establishing their post-conditions, methods must also ensure that state invariants are satisfied. Ensuring that weaving does not cause violations of state invariants is another difficult challenge for aspect developers, and another source of errors.

7.2.5 Incorrect Focus of Control Flow

A pointcut designator selects which of a method’s join points to capture. This selection is determined at weave time. However, there are often cases where the information
needed to correctly make such a decision is available only at run time (in languages that support dynamic weaving, such as AspectJ). Sometimes join points should only be selected in a particular execution context. This context could be within the control structure of a particular object, or within the control flow that occurs below a point in the execution.

7.2.6 Incorrect Changes in Control Dependencies

The advice type "around" can alter the behavioral semantics of a method upon which it is applied. New code is inserted, new branches appear that alter the dependencies among statements, and new data may also be inserted. These faults can occur when for example, an around advice and a matched join-point have different control flows. These faults will affect core concern behavior. In [5], they propose to apply regression testing strategies in this case.

7.3 Aspect Prevention Mechanism

The best strategy of handling conflicts is to prevent conflicts from happening. Conflicts can be prevented by adding arbitrators to arbitrate the conflict between components or seeking alternative components, based on the causes of conflicts. We will define a model for the prevention policy of each of the fault types presented in the previous section. The collection of these models represents our prevention mechanism for a given aspect application.

7.3.1 Model of Incorrect Aspect Precedence

We use the example shown in Figure 7.2 to describe how to define the prevention policy for this kind of fault in aspect-oriented systems. This example is taken from [63]. The example consists of a simple personnel management system. The Employee class forms an important part of the system. In particular, we will focus on the method increaseSalary(), which uses its argument to compute a new salary. This example has been constructed as a scenario that introduces new requirements at each step. Applying the principle of separation of concerns, each of these requirements is represented by aspects that will be woven on the same join point (as well as others), after the execution of the method increaseSalary() of the Employee class.
As a first step, the company introduces a logging system to monitor the change of salaries. This feature is represented by the `MonitorSalary` aspect. This aspect prints a notification whenever a salary has been changed. This could include information about the employee and the type of salary change.

The second requirement states that certain classes of the application should store their state in a database. The database should be updated, as soon as possible, after each state changes in the object. To keep persistence separate from the application model, an aspect is used to realize this requirement. The `PersistenceProtocol` aspect contains the advice that performs the update of a persistent object. If the data of a persistent object changes, the corresponding information should be updated in the database too.

Because the database needs to be updated as soon as possible after the state changes in the object, the advice of the `PersistenceProtocol` aspect has to be executed before the advice of the `MonitorSalary` aspect.

The next requirement states that an employee’s salary cannot be higher than his/her manager’s salary. Thus, a raise is not accepted if it violates this criterion. This is enforced by the `CheckRaise` aspect. The advice of this aspect checks the new salary after the `increaseSalary()` method has executed. If the rule is violated, a warning message is
printed and the salary is set back to its original value.

Adding the CheckRaise aspect affects the composition; if this aspect fails the PersistenceProtocol aspect should not be executed because the employee’s data has not changed. That is, the execution of the PersistenceProtocol aspects depends on the outcome of CheckRaise.

The fourth requirement states that if the database is not available, persistence must be implemented with XML files. For each instance of Employee, an XML file is generated. If the regular persistence does not take place (e.g., because of database connection problems), the file must be updated after each state change of the Employee object. This is realized by the UpdateXML aspect. This aspect has one advice that calls the method that rewrites the XML file if the salary (or other data) changes.

In this example, XML files should be updated only if the PersistenceProtocol aspect was not able to update the database. This means that UpdateXML should also execute conditionally; only if PersistenceProtocol failed. The execution of an aspect may depend on the outcome of other aspects. Only if the outcomes of these other aspects satisfy a certain criterion is the dependent aspect allowed to execute.

To define the prevention policy of this fault type, we can follow these three steps:

1. Define the aspect precedence requirements or constraints for a given application.

2. Formalize each constraint by using the logic $L_A$.

3. Define a coordination aspect module which will contains mainly these formalized constraints under the field Prescription Axioms. This field defines properties describing what the system should do, or proscribing the violation of desired properties, i.e., undesirable interactions. It is a field of our component abstraction (see section 5.2 of chapter 5).

7.3.1.1 Step 1

The constraints of the management system of the section 7.2.1 are: (1) Persistence-Protocol aspect should be executed before MonitorSalary aspect. (2) CheckRaise aspect should be executed before PersistenceProtocol aspect. If CheckRaise aspect returns an error message (i.e., the method check returns an error message), PersistenceProtocol aspect should not be executed (that is, it is forbidden to perform update action). (3) If PersistenceProtocol aspect is not executed, then UpdateXML aspect should be performed.
7.3.1.2 Step 2

The formalization of the constraints are done on instances of the aspects and classes. Let em, mo, pe, ch, up, be instances of Employee class, MonitorSalary, PersistentProtocol, CheckRaise, UpdateXML aspects, respectively. x, y are variables of sort int, s, s’ are variables of sort string, and f is a variable of sort file. A sort is like a type in a programming language. The translation of the first constraint into the logic $L_A$ is as follows:

$$(Afpr(salaryChange(em.salary))) (I(mo.print(s) ; pe.update \lor mo.print(s) ; pe.print(s'))$$

This formula means that each time the join point call(em.increaseSalary(int)) is selected by the pointcut salaryChange(em.salary) and immediately after this join point (the part $Afpr(salaryChange(em.salary)))$, we should never perform the action mo.print(s);pe.update or mo.print(s);pe.print(s’). Recall that I is the symbol of the forbidden operator of the logic $L_A$ and ; the one of the sequential actions composition operator. The translation of the second constraint into the logic $L_A$ is as follows:

$$(Afpr(salaryChange(em.salary))) (I(pe.update; ch.check(x, y) \lor pe.print(s'); ch.check(x, y)) \land (ch.check(x, y) = error) \rightarrow I(pe.update \lor pe.print(s'))$$

This formula means that each time the join point call(em.increaseSalary(int)) is selected by the pointcut salaryChange(em.salary) and immediately after this join point, it is forbidden to perform the action pe.update;ch.check(x,y) or pe.print(s’);ch.check(x,y), and if the check method returns an error message, then it is forbidden to execute the action pe.update or pe.print(s’). The translation of the third constraint into the logic $L_A$ is as follows:

$$(Afpr(salaryChange(em.salary))) ([\neg (pe.update \land pe.print(s'))] up.updateFile(f))$$

This formula means that each time the join point call(em.increaseSalary(int)) is selected by the pointcut salaryChange(em.salary) and immediately after this join point, if the action $pe.update \land pe.print(s')$ is not executed, then perform the action up.updateFile(f). Note that $[\cdot]$ is the necessity operator of the logic $L_A$. 
7.3.1.3 Step 3

The coordination module aspect will mainly contain the above properties under the field Prescription Axioms. It records information about all aspect modules which are present in the application and also the classes. It can contain other elements such as mechanisms that add and remove dynamically an aspect to and from the application, respectively. The set of its advices contains at least the action $T^?$, an action which changes nothing. This coordination aspect module is responsible for the management of the aspect scheduling or ordering at the shared join points. This module concept reinforces the separation of concerns principle of aspect technology and therefore improves the modularity principle.

7.3.2 Model of "Incorrect Strength in Pointcut Patterns"

We assume that the specifier has correctly determine the joinpoints with regard to each advice and the related base component. For a pointcut $PC = \{jp_i, i \in \{1, 2, ..., n\}\}$ related to a base component BP and advice $\alpha$ of an aspect $A$, we can state the following lines:

1. if $jp \notin PC$ (i.e., if the joinpoint $jp$ is not in the pointcut) or $\neg pr(jp)$ (i.e., if $jp$ has not been selected), it is forbidden to perform action $\alpha$ at $jp$. This sentence avoids the cases of superset or overset selection of joinpoints (parts A, B, C of figure 7.1).

2. The formulae $(B,pr(jp))\alpha, (A,pr(jp))\alpha, (I,pr(jp))\alpha$ and $(A,pr(jp))\alpha$ (see section 4.3 of chapter 4) avoid the cases of subset or underset selection of joinpoints (part D of figure 7.1).

In the following, we use the abbreviations:

- $(B,pr(jp))\alpha$ for $G(\neg pr(jp) \rightarrow I(\alpha;jf(jp);T^?)/T^?)$ Before advice case.
- $(A,pr(jp))\alpha$ for $G(\neg pr(jp) \rightarrow I(fj(jp); \alpha;T^?)/T^?)$ After advice case.
- $(I,pr(jp))\alpha$ for $G(\neg pr(jp) \rightarrow I((fj(jp)); \alpha;T^?)/T^?)$ Instead Of advice case.

The formulae $(B,pr(jp))\alpha, (A,pr(jp))\alpha$, and $(I,pr(jp))\alpha$ mean the above point 1. In other words, $(B,pr(jp))\alpha$ means that in any state, if $jp$ is not selected then, it is forbidden to perform action $\alpha$ followed by the action encapsulated in the body of the joinpoint


\[ (A_{pr}(jp))\alpha \] means that in any state, if \( jp \) is not selected then, it is forbidden to perform action encapsulated in the body of the joinpoint \( jp \) followed by the action \( \alpha \).

\[ (I_{pr}(jp))\alpha \] means that in any state, if \( jp \) is not selected then, it is forbidden to replace action encapsulated in the body of the joinpoint \( jp \) by the action \( \alpha \).

The prevention policy of Incorrect Strength in Pointcut Patterns fault type will consist of the following formulae:

- \((B_{epr}(jp_i))\alpha \) and \((B_{fpr}(jp_i))\alpha \) with \( i \in K_1 \subseteq \{1, 2, ..., n\} \) for before advice case; \( K_1 \) is the set of the joinpoints related to the before advice case.

- \((A_{fpr}(jp_i))\alpha \) and \((A_{fpr}(jp_i))\alpha \) with \( i \in K_2 \subseteq \{1, 2, ..., n\} \) for after advice case; \( K_2 \) is the set of the joinpoints related to the after advice case.

- \((I_{epr}(jp_i))\alpha \) and \((I_{fpr}(jp_i))\alpha \) with \( i \in K_3 \subseteq \{1, 2, ..., n\} \) for InsteadOf advice case; \( K_3 \) is the set of the joinpoints related to the instead advice case.

- \((A_{pr}(jp_i))\alpha_1 \alpha_2 \) and \((B_{pr}(jp_i))\alpha_1 \) and \((A_{pr}(jp_i))\alpha_2 \) with \( i \in K_4 \subseteq \{1, 2, ..., n\} \) for around advice case. \( K_4 \) is the set of the joinpoints related to the around advice case.

Note that \( K_i \) may be empty for some \( i \). Since \((B_{epr}(jp_i))\alpha \), \((A_{fpr}(jp_i))\alpha \), \((I_{epr}(jp_i))\alpha \) and \((A_{pr}(jp_i))\alpha \) are already in the aspect, according to the case, we don’t need to put them in the coordination aspect. Thus, the prevention policy of the Incorrect Strength in Pointcut Patterns fault type will be the following set of formulae in the coordination aspect module under the Prescription Axioms field:

\[
(\bigcup_{i \in K_1}\{B_{fpr}(jp_i)\alpha\}) \\
\cup (\bigcup_{i \in K_2}\{A_{fpr}(jp_i)\alpha\}) \\
\cup (\bigcup_{i \in K_3}\{I_{fpr}(jp_i)\alpha\}) \\
\cup (\bigcup_{i \in K_4}\{B_{fpr}(jp_i)\alpha_1 \land (A_{fpr}(jp_i)\alpha_2)\})
\]

The meaning of each formula of this set is defined in the above lines.

### 7.3.3 Model of "Failure to Establish Expected Post-Conditions"

Method preconditions are assertions that must hold when the execution of the method starts. Method postconditions are assertions that must hold immediately after the execution of the method ends. To prevent the occurrence of this fault type in an aspect oriented
system, we have to analyze the effect of the execution of the advice on the context of the client class, i.e., the effect of the advice body at each selected joinpoint of the client class.

7.3.3.1 Representation of an Action

We can represent an action (a program) \( \alpha \) as Hoare clauses, i.e., \( \{ \varphi \} \alpha \{ \psi \} \), where \( \varphi = \text{Pre}(\alpha) \) is the precondition of \( \alpha \) and \( \psi = \text{Post}(\alpha) \) its postcondition. If \( \alpha \) does not have any precondition, then \( \alpha \) can be written \( \{ \top \} \alpha \{ \psi \} \) where \( \top \) is the boolean true value. In our logic \( L_A \), \( \{ \varphi \} \alpha \{ \psi \} \) can be represented by \( \varphi \rightarrow [\alpha] \psi \) or \( \varphi \rightarrow (\lfloor \alpha \rfloor O(\psi))/\top \).

7.3.3.2 Representation of a Joinpoint

A joinpoint \( jp \) can be represented by \( \{ \varphi \} f j(jp) \{ \psi \} \) where \( f j(jp) \) is the action encapsulated in \( jp \), \( \varphi \) is the precondition of this action, and \( \psi \) its postcondition. Thus, in \( L_A \), \( jp \) can be represented by \( \varphi \rightarrow ([f j(jp)]O(\psi))/\top \).

7.3.3.3 Prevention Mechanism

In our analysis, we distinguish four cases: one case for each advice type (Before, After, Around, InsteadOf). We notice that the reasoning is the same for all these four cases. In this section, we show this analysis with Before advice. Before advice is written in the logic \( L_A \) as \( (B_{\text{pr}}(np))\beta \) where \( np \) is a pointcut expression, \( \beta \) is the body of the advice. Assume that \( np = jp_1 ||jp_2 \) (we can do the same thing if \( np \) contains an arbitrary number \( n \) of joinpoints). Let \( jp_i \) represented by \( \varphi_i \rightarrow ([f j(jp_i)]O(\psi_i))/\top \), \( i \in \{1, 2\} \). We know that

\[
(B_{\text{pr}}(np))\beta \equiv (B_{\text{pr}}(jp_1))\beta \lor (B_{\text{pr}}(jp_2))\beta \text{ and } (B_{\text{pr}}(jp_i))\beta \equiv G(\text{pr}(jp_i) \rightarrow O(\beta; f j(jp_i); T?)/T?).
\]

The formula \( (B_{\text{pr}}(jp_i))\beta \rightarrow \varphi_i \) is always true because in \( (B_{\text{pr}}(jp_i))\beta \), we have \( O(\beta; f j(jp_i); T?) \). This means that the execution of \( \beta \) does not avoid the execution \( f j(jp_i) \). Thus, the preconditions \( \varphi_i \) of \( f j(jp_i) \) is satisfied. To guarantee that the postcondition \( \psi_i \) of the action \( f j(jp_i) \) will always satisfied, it suffices to add the following formula
into the coordination aspect module under the Prescription Axioms field:

\[ \phi_i \rightarrow ([f(jp_i)]O(\psi_i))/\top, \quad \text{for } i \in \{1, 2\}. \]

which means that after each time the action \( f(jp_i) \) is performed, the postcondition \( \psi_i \) must be satisfied. If \( np = jp_1 \& \& jp_2 \), the reasoning is the same as in the case where \( np = jp_1 || jp_2 \). Thus, the prevention policy of the Failure to establish expected postconditions fault type will be the set of these following formulae in the coordination aspect module under the Prescription Axioms field:

\[ \phi_i \rightarrow ([f(jp_i)]O(\psi_i))/\top \]

for all joinpoints \( jp_i \) in a aspect oriented system.

### 7.3.4 Model of "Failure to Preserve State Invariants"

Generally speaking, the aspects of crosscutting concerns alter the control flows defined in the core concern. From the perspective of state models of system behavior, they not only modify the state-transition relations but also possibly introduce extra states in the state models of objects defined by their classes. This can affect or even change the behavior of objects specified by the base programs [114]. State invariants are formulas that are true in all states of a correct execution. The prevention policy for this fault type is a set of the following formulae in the coordination aspect module under the Prescription Axioms field:

\[ GO(p?) \]

for all state invariants \( p \) in a aspect oriented system. \( G \) is the always modality of our logic and \( O \) the obligation modality. \( GO(p?) \) means that the invariant \( p \) must hold in all states.

### 7.3.5 Examples

We will use two examples to illustrate our prevention mechanism. The first one is for Incorrect Strength in Pointcut Patterns fault type and the second one is for other fault types for which we defined a prevention mechanism in this chapter. The first example shows that our approach can be applied to a real application; while the second one (even though it is a simple example) is adequate for these fault types. The first example is taken
from [113]. This example consists of a greeting card purchase subsystem of an online shopping application. The subsystem consists of 12 classes, 3 aspects and 398 lines of code. The application supports flexible business rules for a variety of greeting cards. The price of a card in a purchase depends on the card type and the quantity of the cards. The cards are categorized as holiday cards, birthday cards, valentine cards, etc. Different types of cards have different unit price, minimum quantity of cards for discount, and discounted unit price. Discounted unit price is available only when the quantity is greater than the required minimum. There are numerous base classes (core functional concerns) in this example: Card, Price, HolidayCardPrice, BirthdayCardPrice, ValentineCardPrice, Purchase, Transaction; but we consider only two classes Card and Purchase to illustrate the prevention model of Incorrect Strength in Pointcut Patterns fault type. A card has information such as code and price. A purchase includes a particular type of card and the quantity of the cards. The aspect we consider here is AccessControl aspect. This aspect is responsible for distinguishing legal and illegal access to sensitive data, such as price information. A password-based access control has been used for security check. Figure 7.3 shows a class/aspect diagram with the two classes: Purchase and Card. Each class has a join point collected by pointcut expression priceMonitor in aspect AccessControl, which facilitates communication between Purchase and Card objects. The priceMonitor pointcut is associated with an InsteadOf advice which checks if the caller is authorized. If so, the caller may proceed to invoke getPrice; otherwise the access is denied. Thus,

\[ \text{advice action} = \alpha \lor \beta, \] 

\[ \alpha = [\text{authorize}?]\text{getPrice} \] 

\[ \beta = [(\neg\text{authorize})?]\text{denyAccess} \] 

\( \alpha \) is the action that executes getPrice each time the caller is authorized. denyAccess is the action that denies access to the caller. \( \beta \) is the action that denies access to the caller each time he is not authorized. The prevention policy related to this fault type for this subsystem depicted in the figure 7.3 consists of the following prescription axiom defined in a coordination aspect module:

\[ (I_{ipr}(\text{withincode}(\ast \text{Purchase.getCharge()}) \&\& \text{call}(\ast \text{Card.getPrice()}))) \alpha \lor \beta \] (7.1)
since,

\[(L_{o}pr(\text{withincode}(* \text{Purchase.getCharge()} && \text{call(* Card.getPrice())})) \vee \beta) \] (7.2)

will be defined in aspect AccessControl. Formula 1 says that in any state, if the joinpoint \text{withincode}(* \text{Purchase.getCharge()} && \text{call(* Card.getPrice())}) is not selected then, it is forbidden to replace action encapsulated in the body of this joinpoint by the action \(\alpha \vee \beta\). The second formula is an instead advice axiom. It means that in each state \(s\) where the joinpoint \text{withincode}(* \text{Purchase.getCharge()} && \text{call(* Card.getPrice())}) is selected, the action \(\alpha \vee \beta\) is obliged to be executed instead of the action (encapsulated in the body of this joinpoint) that should be executed in \(s\).

We use the example shown in Figure 7.2 to describe how to define the prevention policy for \textit{Failure to Establish Expected Post-Conditions} and \textit{Failure to Preserve State Invariants} fault types in aspect-oriented systems. We have already used this example to illustrate the model of aspect precedence fault type. We adapt this example for this section purpose. The method \textit{increaseSalary()} uses its argument to compute a new salary.
The precondition $\varphi$ and postcondition $\psi$ of this method are:

$$\varphi : \begin{cases} \text{the employee has worked one more year} \\ \end{cases}$$

$$\psi : \begin{cases} - new_{\text{Salary}} = old_{\text{Salary}} + \Delta_{\text{Salary}} \\ - \text{the employee is happy} \\ \end{cases}$$

We are interested in the aspect $ManageBonus$. This aspect manages attributions of the bonus and penalties to the employees. If the performance of an employee is higher than a threshold value $t_1$, $ManageBonus$ adds a $+$-bonus to the salary of this employee. But, if the employee performance is less than a threshold valuer $t_2$, $ManageBonus$ adds a $-$-bonus to the salary of this employee. The only advice method of the aspect $ManageBonus$ which, is applied before the joinpoint $\text{execution(Employee.increaseSalary(int))}$, is $addBonus(int b)$. The aspect $ManageBonus$ may break down the postcondition of the joinpoint body in a case an employee will be unhappy because his performance is less than the threshold valuer $t_2$. Besides, Employee class has a state invariant $p$

$$p : \begin{cases} \text{an employee’s salary cannot be higher than his/her} \\ \text{manager’s salary} \\ \end{cases}$$

The aspect $ManageBonus$ may break down this invariant in a case the new salary of the employee becomes greater than his manager’s salary. The prevention policy related to these two aspect fault types for this personal management system consists of the following prescription axioms defined in a coordination aspect module:

$$\varphi \rightarrow ([fj(\text{execution(em.increaseSalary(int))})]O(\psi)) / \top? \quad (7.3)$$

and

$$GO(p?) \quad (7.4)$$

$em$ is an instance of Employee class. Formula 7.3 says that after each execution of the method $\text{increaseSalary(int)}$, the postcondition $\psi$ must be satisfied. Formula 7.4 says that the invariant $p$ must always be satisfied.
7.4 Verification

Concerning the verification, proving that the specified object (module) has a certain property consists in proving that the property is a logical consequence of the axioms of the specification. The aim of a modular verification is to provide the possibility of reusing properties of components when the specifications of the components are themselves reused when building larger systems. The strategy is to prove locally a property at the level of the specification of a component and use it as lemma when reasoning about global properties of a system.

In the case of our prevention properties, it is a very simple importation because the property is an axiom. Indeed, suppose for instance that we want to verify that at any time, the constraints described in sections 7.3.1, 7.3.3, 7.3.4 are satisfied in our personnel management system. These constraints are specified in the coordination aspect, under the field Prescription Axioms. To guaranty that these formulae are not violated in the system, it suffices to weave at first (before weaving any other aspect) the coordination aspect into the base module Employee. By weaving the coordination aspect into the base module Employee, all prescription axioms of this coordination Aspect become theorems of the modular system, by the weaving operator which is basically based on the category theory colimit concept. Recall that this notion of colimit satisfied the following merging requirements:

1. **Non-redundancy**: Only one copy of the common parts is included in the merged model.

2. **Completeness**: Merge should not lose information, i.e., it should represent all the source models completely.

3. **Minimality**: Merge should not introduce information that is not present in or implied by the source models.

4. **Semantic Preservation**: Merge should support the expression and preservation of semantic properties.

Thus, if we want to verify an invariant $p$ is satisfied, we just need to check that the relatives prescription axioms, i.e., $GO(p?)$ is a theorem in the modular system. The presence of these prescription axioms provides means to regulate the societal life of aspect and classes' actions (or methods) in the modular system environment. The principle of this
modular verification is as follows: for a morphism \( m : \text{MOD}_1 \rightarrow \text{MOD}_2 \), if a property \( P \) is true in \( \text{MOD}_1 \), then \( m(P) \) is true in \( \text{MOD}_2 \). For example, if we want to prove a property (by model-checking or a theorem proving technique) on the module ModSystem representing the entire system and we have the configuration in figure 1.1 of the introduction 6.1. This property may be expressed and proved in Mod3, and then translated to ModSystem. Another property of ModSystem may result from the conjunction of two lemmas, each lemma being provable in smallest module (Mod3 and Mod4). It is interesting to exploit the structure of specifications to verify properties in a modular way and it is easier to prove a property on a smaller specification that on a larger one. Order between specification is induced by the morphisms used to interconnect them: \( \text{SPEC}_1 \) is smaller than \( \text{SPEC}_2 \) if there exist a morphism \( m : \text{SPEC}_1 \rightarrow \text{SPEC}_2 \). It is worth to note that it is sometimes impossible to prove a property when state space is too big.

By using category theory, we can take advantage of the structure of a system specification to do the verification task and to infer some desirable properties on the global system. We believe that our approach can provide an important contribution to control reasoning over large systems once these systems have been designed along methods which, like aspect-orientation, favor modularity and reusability.

### 7.5 Conclusion and Future Work

In this chapter, we presented a prevention mechanism for four aspect fault types that can cause undesirable behaviors in an aspect-oriented system. These fault types are: Incorrect aspect precedence, Incorrect strength in pointcut patterns, Failure to establish expected post-conditions and Failure to preserve state invariants. We are currently working on the specification in the logic \( \mathcal{L}_A \) of all other aspect fault types presented above.

Most of the aspect-oriented verification approaches are based on a detection and correction strategy, contrary to our approach which is based firstly on a prevention strategy and then on a correction strategy if it is necessary. The contribution of this chapter is the prevention policy that we develop for the fault types Incorrect aspect precedence, Incorrect strength in pointcut patterns, Failure to establish expected post-conditions and Failure to preserve state invariants. The deontic concepts enforce this prevention policy.

The best strategy of handling conflicts is to prevent conflicts from happening. We can convince ourselves by making an analogy to the medicine, where experts and governments prefer to place more emphasis on measures to prevent disease. The same analogy
could be observed in avionics where MRO (maintenance, repair, overhaul) activities put more emphasis on preventive actions than correctives actions for flight quality and reliable aircraft. The spirit of our work is similar to one of Intrusion Prevention Systems.

For future work, we plan to extend to aspect concepts the tool MOKA developed by the team of Wielo [61, 110] to do some verification tasks. Currently, this tool is in standard ML. We need to rebuild it in an object oriented language such as OCAML, by integrating aspect concepts or we can also use an aspect oriented language such as Aspectsual CAML [56], because our approach is aspect-oriented. We think that the development of this tool will take a lot of time.
CHAPTER 8

CASE STUDY

In this chapter, we illustrate our approach by means of an industrial case study. This case study entitled "NextGen MRO Instructions Application" has been proposed by Pratt & Whitney Canada [2]. It is a project on which we worked during a training program we did in this aerospace company in summer 2011. This case study shows that our approach can be applied to a real application. We used object-oriented technology to develop this web application for Pratt & Whitney Canada. Is is a 5-tiers architecture application we developed during the training program [81]. The objective of this chapter is not to present here all the work we did for Pratt & Whitney Canada, but to illustrate our formal framework using this project. In our approach, we used aspect-orientation, category theory and algebraic specification techniques to formally specify softwares. The benefits here by applying our approach to this Pratt’s project are the formal specification and verification, which allows modular reasoning, and aspect technology advantages. MRO stands for Maintenance, Repair, Overhaul. In the future, we can propose to Pratt & Whitney Canada to refactor this application by aspect-oriented technology.

8.1 Project Description

According to [1], MRO may be defined as, "All actions which have the objective of retaining or restoring an item to a state in which it can perform its required function. The actions include the combination of all technical and corresponding administrative, managerial, and supervision actions". In many organizations because of the number of devices or products that need to be maintained or the complexity of systems, there is a need to manage the information with software packages. This is particularly the case in aerospace (e.g. airline fleets), military installations, large plants (e.g. manufacturing, power generation, petrochemical) and ships. These software tools help engineers and technicians in increasing the system availability and reducing costs and repair times as well as reducing material supply time and increasing material availability by improving supply chain communication. Current methods of managing MRO are limited due to their inherent legacy structures, restricted visibility and reactive mode (in sense of reactive systems) of operation, leading to excessive operating costs. Such costs account for lost revenue when measured in term of aircraft downtime [60].
The goal of this project is to develop and deploy an interactive AIS Viewer for use in Pratt & Whitney Canada overhaul shops. This tool should generate and maintain work scopes from the existing technical manuals (AIS stands for Assembly Instructions Sheet). The NextGen MRO web application is intended to provide a breakthrough in the delivery, data capture and integration of aircraft engine information. Using the technical content from the maintenance and repair manuals and the parts information, the NextGen MRO application will step by step assist the mechanics and the inspectors to maintain the engines. This project is unique as the application re-purposes and re-packages the technical data from the published manuals. The user can tailor the instructions to the specific work scope for each engine configuration. The objective of this project is to develop an e-enable MRO system that helps to manage operations in a safe as well as cost-effective manner. The MRO system will provide benefits to the company and contribute to its long term strategy of maintaining a competitive market presence. By using the web-based solution, the company will be in a better position to optimize its maintenance and overhaul and repair activities [60]. The proposed solution will provide benefits to the company, its employees, as well as its partners and suppliers. The system should also improve current business processes by removing non-value adding activities that are currently present. It is worth to note that this e-MRO system is a task optimization system. In total, this e-MRO system will allow to move from the traditional paper-based methods to a more advanced electronic format method. Such method is based on the notion of improving visibility through the incorporation of next generation Internet technologies. In order to remain highly competitive in airline operations area, MRO facilities need to develop new methods of operating more effectively whilst incorporating the latest advances in Internet tools and information technologies, which would facilitate the global sharing of related MRO data. In addition, applying aspect technology to this e-MRO application will bring into this later all aspect benefits. To the best of our knowledge, there is not yet in the literature any aspect orientation of MRO systems.

8.2 Partial Modular Specification of the MRO System

This section presents a specification of some parts of the e-enable MRO system. We think that it suffices to illustrate the main concepts of our modular approach. At requirement analysis, we used use case technique to identify, clarify and organize system requirements. Figure 8.1 shows the use case diagram of the MRO application. The primary actors (the principal actors that call upon system services to fulfill a goal) are:
Mechanic, Inspector, AHS Editor, Control Document Editor, AISBuilder, AISWebsite. The secondary actors are: Server, Sap System, Contenta, Documentum, Administrator, Technical Editor. Tables 8.I and 8.II present activities for each actor. Tables 8.III, 8.IV and 8.V present use case descriptions without too many details. Figure 8.2 shows the class diagram of the MRO application.
Figure 8.1: Use Case Model of MRO Application.
<table>
<thead>
<tr>
<th>Actor</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mechanic</td>
<td>- Read and follow instructions in order to assembly and dis-assembly items.</td>
</tr>
<tr>
<td></td>
<td>- Record it progress, information.</td>
</tr>
<tr>
<td></td>
<td>- Check if tasks are well done.</td>
</tr>
<tr>
<td></td>
<td>- Order items.</td>
</tr>
<tr>
<td></td>
<td>- Call mechanic or inspector for checking.</td>
</tr>
<tr>
<td>Foreman</td>
<td>- Assigns tasks (scopes) to Mechanics.</td>
</tr>
<tr>
<td>LineTech</td>
<td>- Create scopes from OHM to Foremen.</td>
</tr>
<tr>
<td></td>
<td>- Give instructions to mechanics via Foramen.</td>
</tr>
<tr>
<td>Contenta</td>
<td>- Provide manuals to AIS system in SGML mark-up called &quot;snapshots&quot; (one file per manual)</td>
</tr>
<tr>
<td>Documentum</td>
<td>- Provides graphics to be converted by the AISBuilder.</td>
</tr>
<tr>
<td>SAPSystem</td>
<td>- Provide sale orders information, additional information about engines.</td>
</tr>
<tr>
<td></td>
<td>- Build requirements.</td>
</tr>
<tr>
<td></td>
<td>- Order items.</td>
</tr>
<tr>
<td></td>
<td>- Record data.</td>
</tr>
<tr>
<td>AHS Editor</td>
<td>- Specify and edit witness requirements.</td>
</tr>
<tr>
<td></td>
<td>- Add special instructions to manuals.</td>
</tr>
<tr>
<td></td>
<td>- Add recording requirements.</td>
</tr>
<tr>
<td></td>
<td>- Calculate.</td>
</tr>
<tr>
<td></td>
<td>- Read.</td>
</tr>
</tbody>
</table>

Table 8.1: Actor activities: Part 1
<table>
<thead>
<tr>
<th>Actor</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspector</td>
<td>- Check or ensure that instructions have been followed correctly.</td>
</tr>
<tr>
<td></td>
<td>- Read assembly and disassembly instructions.</td>
</tr>
<tr>
<td></td>
<td>- Record validation results.</td>
</tr>
<tr>
<td></td>
<td>- Order items.</td>
</tr>
<tr>
<td>Technical editor</td>
<td>- Read comments (in order to update manuals in contenta; this is not about AISViewer).</td>
</tr>
<tr>
<td>Administrator</td>
<td>- Maintain AIS system.</td>
</tr>
<tr>
<td></td>
<td>- Manage access and privilege rights.</td>
</tr>
<tr>
<td></td>
<td>- Read.</td>
</tr>
<tr>
<td></td>
<td>- Write.</td>
</tr>
<tr>
<td>Server</td>
<td>- Collect user inputs.</td>
</tr>
<tr>
<td></td>
<td>- Generate reports.</td>
</tr>
<tr>
<td></td>
<td>- Support AHS editors.</td>
</tr>
<tr>
<td></td>
<td>- Feedback user inputs.</td>
</tr>
<tr>
<td>Control Document</td>
<td>- Coordinate technical reviews.</td>
</tr>
<tr>
<td>Creator</td>
<td>- Validate instructions.</td>
</tr>
<tr>
<td></td>
<td>- Create AIS structures in XML format called control files.</td>
</tr>
<tr>
<td></td>
<td>- Read.</td>
</tr>
<tr>
<td></td>
<td>- Update.</td>
</tr>
<tr>
<td>AIS Website</td>
<td>- Provide Html, PNG, Javascript, XML data to the navigator.</td>
</tr>
<tr>
<td>AIS Builder</td>
<td>- Create Html, PNG.</td>
</tr>
<tr>
<td></td>
<td>- Provide Html, PNG to AIS Website.</td>
</tr>
</tbody>
</table>

Table 8.II: Actor activities: Part 2
<table>
<thead>
<tr>
<th>Use case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disassemble</td>
<td>- The mechanic logs to MRO system.</td>
</tr>
<tr>
<td></td>
<td>- He chooses a sale order, AIS or TIS.</td>
</tr>
<tr>
<td></td>
<td>- He reads instructions.</td>
</tr>
<tr>
<td></td>
<td>- He follows instructions.</td>
</tr>
<tr>
<td></td>
<td>- He tears down motors.</td>
</tr>
<tr>
<td>Assemble</td>
<td>- The mechanic logs to MRO system.</td>
</tr>
<tr>
<td></td>
<td>- He chooses a sale order, AIS or TIS.</td>
</tr>
<tr>
<td></td>
<td>- He reads instructions.</td>
</tr>
<tr>
<td></td>
<td>- He follows instructions.</td>
</tr>
<tr>
<td></td>
<td>- He updates operations/sequence status (e.g., ND, NA, D...) and adds comments for each marked status.</td>
</tr>
<tr>
<td>Read</td>
<td>- The actor logs to MRO system.</td>
</tr>
<tr>
<td></td>
<td>- He open the file.</td>
</tr>
<tr>
<td>Record</td>
<td>- The actor logs to MRO system.</td>
</tr>
<tr>
<td></td>
<td>- He selects the build mode.</td>
</tr>
<tr>
<td></td>
<td>- He selects a sale order and a build number.</td>
</tr>
<tr>
<td></td>
<td>- He inserts data.</td>
</tr>
<tr>
<td>Check</td>
<td>- Abstract use case that is extended by Post-Check and Live-Check.</td>
</tr>
<tr>
<td>Post-Check</td>
<td>- The Mechanic/Inspector logs to MRO system.</td>
</tr>
<tr>
<td></td>
<td>- He reads instructions.</td>
</tr>
<tr>
<td></td>
<td>- He follows instructions.</td>
</tr>
<tr>
<td></td>
<td>- He verifies what the mechanic did.</td>
</tr>
<tr>
<td></td>
<td>- He signs off.</td>
</tr>
<tr>
<td></td>
<td>- He closes file.</td>
</tr>
</tbody>
</table>

Table 8.III: Use Case Description: Part 1
<table>
<thead>
<tr>
<th>Use case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCheck</td>
<td>- Sign &quot;C-check&quot;.</td>
</tr>
<tr>
<td>WCheck</td>
<td>- Sign &quot;W-check&quot;.</td>
</tr>
</tbody>
</table>
| Live-Check | - The Inspector reads and follows instructions.  
|           | - He checks what the mechanic is doing.  
|           | - He signs off.  
|           | - He logs off (at second level). |
| Assign   | - Foreman logs to MRO system.  
|           | - He maps work scopes to mechanics. |
| Create Scopes | - Line Tech logs to MRO system.  
|            | - He retires sale orders from SAP.  
|            | - He analyzes sale orders.  
|            | - He gives the first disassembly instructions.  
|            | - He writes a tear down report.  
|            | - He creates work scopes and gives them to foreman. |
| Provide  | - The actor supplies information or data files. |
| Build Requirements | - SAP system specifies what is need to be done on the engine and the build spec requirements; the engine can be repair, overhaul, or converted. |
| Edit     | - The AHS Editor logs to MRO system.  
|           | - He changes status to AHS edit.  
|           | - He updates procedures (adds special instructions supplemental to manuals).  
|           | - He validates.  
|           | - He transfers to production. |
| Write    | - The actor logs to MRO system.  
|           | - He writes on the file. |
| WICheck  | - Sign "W-Check" (Inspector only). |
| CICheck  | - Sign "CI-Check" and change status of a step (NA, ND, Done) and add remarks (CI level only). |

Table 8.IV: Use Case Description: Part 2
<table>
<thead>
<tr>
<th>Use case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage</td>
<td>- The administrator logs to MRO system.</td>
</tr>
<tr>
<td></td>
<td>- He assigns rights to MRO system users.</td>
</tr>
<tr>
<td></td>
<td>- He removes rights to MRO system users.</td>
</tr>
<tr>
<td></td>
<td>- He create rights.</td>
</tr>
<tr>
<td></td>
<td>- He logs off.</td>
</tr>
<tr>
<td>Maintain</td>
<td>- The administrator logs to MRO system.</td>
</tr>
<tr>
<td></td>
<td>- He does different administration tasks.</td>
</tr>
<tr>
<td></td>
<td>- He logs off.</td>
</tr>
<tr>
<td>Support</td>
<td>- The server gets AHS.</td>
</tr>
<tr>
<td></td>
<td>- The server reads instructions from server.</td>
</tr>
<tr>
<td></td>
<td>- The server writes (saves) AHS instructions on server.</td>
</tr>
<tr>
<td>Build Report</td>
<td>- The server puts AHS instructions and templates into XSL-FO.</td>
</tr>
<tr>
<td></td>
<td>- The server creates PDF report.</td>
</tr>
<tr>
<td>Validate</td>
<td>- The actor logs to MRO system.</td>
</tr>
<tr>
<td>Build AIS</td>
<td>- He reads instructions and comments.</td>
</tr>
<tr>
<td></td>
<td>- The AIS builder creates SGML pages for each procedure.</td>
</tr>
<tr>
<td></td>
<td>- He copies SGML procedure pages into the collections, and creates a template for each collection.</td>
</tr>
<tr>
<td></td>
<td>- He creates AIS instructions.</td>
</tr>
</tbody>
</table>

Table 8.V: Use Case Description: Part 3
Figure 8.2: Class Diagram of MRO Application.
8.2.1 Aspect Identification

This section covers the identification of aspects we use to illustrate our approach. Some aspect definitions are based on the patterns defined in [20].

8.2.1.1 Authentication Aspect

Authentication is the process of determining whether someone is who is declared to be. This concern tries to prevent that unauthenticated users have access to some web pages. Authentication is a very important concern for building web applications such MRO systems. One needs a username and a password to access to the main page, which is named "toDoCommands" in the context of our MRO system. This concern crosscut many classes such as Mechanic, Administrator, AHSEditor, etc. Thus, similar to the AOSD literature, we represent this authentication concern as an aspect. We name this aspect AuthenticationAspect. This aspect has role to control the access to the MRO system. If the user isn’t authenticated, then he will be redirected to the login page by printing a message to the user. This behavior is defined by means of an action. When a user requests an access to the "toDoCommands" web page by providing his username and password, if these username and password are valid, the user gets access by the operation "getAccess()", otherwise, the access is denied by redirecting the user to the login page; this is done by the operation or action "denyAccess()". Figure 8.3 shows the static view of AuthenticationAspect.

![AuthenticationAspect]

Figure 8.3: Static View of the Authentication Aspect.

Operations of this aspect are:

- getUser(un: string, pw: string): a method that get the user’s information.
- verifyUser(): this method verify if the username and password information provided by the user is valid or not. If it is valid, the method will return true; if not, it will return false.
• **getAccess():** this method uses verifyUser() method to give access to the user in case the user information is valid.

• **denyAccess():** this method uses verifyUser() method to deny access to the user in case the user information is not valid.

• **controlAccess():** this method uses getAccess() and denyAccess() methods to control the user’s access to MRO system.

We choose to use Mechanic class as base class for this aspect for sake of simplicity. All methods of this base class constitute a joinpoint. Table 8.VI shows the weaving detail information.

<table>
<thead>
<tr>
<th>Joinpoints</th>
<th>Pointcuts</th>
<th>Authentication Aspect Operations</th>
<th>Crosscutting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mechanic.*(..)</td>
<td>MROAccessP(): execution (Mechanic.*( .. ))</td>
<td>controlAccess()</td>
<td>Type: before</td>
</tr>
</tbody>
</table>

Table 8.VI: Authentication Aspect Weaving Information for Mechanic and AISDatabase

### 8.2.1.2 Data Origin Authentication Aspect

Data origin authentication is a security service that verifies the identity of a system entity that is claimed to be the original source of received data. A digital signature mechanism can be used to provide this service, as someone who does not know the private key cannot forge the correct signature. Then by using the signer’s public key, anyone can verify the origin of correctly signed data [20]. Data origin authentication is an important security service, where the source of received data can be verified. Authentication service and Data origin authentication service are addressed by aspects since they crosscut many other components.

The definition of Data origin authentication aspect, named DOAAspect, is based on the pattern defined in [20]. Attributes in this aspect include public and private keys the aspect generates, encryption and decryption algorithms used to encrypt and decrypt data, and the encrypted data (digital signatures "SIG"). Several aspect operations are defined to support the use of this aspect and they are referred as aspect Operations:
• generateKey (random-num: int, alg: algorithm): the operation takes a random number and a specified algorithm as parameters. The algorithm is used to generate keys. The output is a key(s) (i.e., public and private keys).

• distributeKey (receiver: String, pubKey: Key): the operation sends a public key to a receiver, where the parameter "receiver" is the address of an actual receiver.

• encryptData (priKey: Key, alg: Algorithm, dt: Data): the operation encrypts data with the private key and the specified algorithm.

• decryptData (pubKey: Key, alg : Algorithm, dt : Data): the operation decrypts data with the public key and the specified algorithm.

• retrieveSIG (data: String): the operation retrieves generated digital signatures for the specified data. The outputs are a collection of digital signatures.

DOAAspect crosscut AISDatabase, SAPSystem, Contenta (OracleDB), Mechanic, Inspector, LineTech, etc. However, we will only use AISDatabase as the sender and Mechanic as the receiver. AISDatabase provides AIS files. AIS stands for Assembly Instruction Sheet.

Figure 8.4 shows Data origin authentication aspect with its two base classes. For AISDatabase class, as its responsibility is to manage AIS data, also the encrypted data need to be stored in the database, therefore, three aspect operations generateKey(), distributeKey(), and encryptData() should crosscut the operation of this class, initialize(). AISDatabase class has also responsibility to search the correct answer for mechanics queries, therefore, its operation search() will be crosscut by the aspect operation retrieveSIG(), as it needs to search the correct digital signatures for the answer data. Before reading the received AIS, a Mechanic interface should decrypt the encrypted AIS data. Thus, its operation read() will be crosscut by the aspect operation decryptData(). The pre-conditions of the operation assemble() of Mechanic class are "sale orders, AIS exist". We note these pre-conditions \( \varphi_a \). The post-condition of the operation assemble() is "status are marked (sign off)". We note this post-condition \( \psi_a \). The pre-condition of the operation disassemble() of Mechanic class is "an engine is present". We note this pre-condition \( \varphi_d \). The post-condition of the operation disassemble() is "the engine is dissociated in a set of pieces". We note this post-condition \( \psi_d \).
Figure 8.4: Data origin authentication aspect with two base classes.

[H] Table 8.VII shows the weaving detail information. An invariant for Data origin authentication aspect is: encryption = decryption. A post-condition of encryptData() operation is encryption = true. A post-condition of decryptData() operation is decryption = true.

<table>
<thead>
<tr>
<th>Joinpoints</th>
<th>Pointcuts</th>
<th>DOA Aspect Operations</th>
<th>Crosscutting</th>
</tr>
</thead>
<tbody>
<tr>
<td>AISDatabase.initialize()</td>
<td>InitializeP(): execution(AISDatabase.initialize())</td>
<td>generateKey(), distributeKey(), encryptData()</td>
<td>Type: after</td>
</tr>
<tr>
<td>AISDatabase.search()</td>
<td>SearchP(): execution(AISDatabase.search())</td>
<td>retrieveSIG()</td>
<td>Type: after</td>
</tr>
<tr>
<td>Mechanic.read()</td>
<td>ReadP(): execution(Mechanic.read())</td>
<td>decryptData()</td>
<td>Type: before</td>
</tr>
</tbody>
</table>

Table 8.VII: Data Origin Authentication Aspect Weaving Information for Mechanic and AISDatabase
8.2.1.3 Role-Based Access Control Aspect

Access control is a very important security concern for our MRO application since Pratt & Whitney manipulates sensitive data. We use the pattern defined in [20] to define the Role-Based Access Control Aspect, which we name RBACAspect. As said in [20], access is the ability to do something with a computer resource (e.g., use, change, or view). Access control is the means by which the ability is explicitly enabled or restricted in some way (usually through physical and system-based controls). Computer-based access controls can prescribe not only who or what process may have access to a specific system resource, but also the type of access that is permitted. Role-based access control (RBAC), as introduced in 1992 by Ferraiolo and Kuhn, has become the predominant model for advanced access control because it reduces the complexity and cost of security administration in large networked applications. The RBAC model is defined in terms of individual users being assigned to roles and permissions being assigned to roles.

The RBAC model includes sets of five basic data elements: users, roles, objects, operations, and permissions. A role is a means for naming many-to-many relationships among individual users and permissions. In addition, the RBAC model includes a set of sessions where each session is a mapping between a user and an activated subset of roles that are assigned to the user. A user is defined as a human being, machines, networks, or intelligent autonomous agents. A role is a job function, within the context of an organization with some associated semantics regarding the authority and responsibility conferred on the user assigned to the role. The process of defining roles should be based on a thorough analysis of how an organization operates and should include input from a wide spectrum of users in an organization. The use of roles to control access can be an effective means for developing and enforcing enterprise-specific security policies, and for streamlining the security management process. Permission is an approval to perform an operation on one or more RBAC protected objects. An operation is an executable image of a program, which upon invocation executes some function for the user [20].

The definition of the RBAC aspect includes the four RBAC concepts as defined in the RBAC security pattern, namely, they are: user, role, object, and right. They are defined as UML classes with attributes and operations (see figure 8.5). The names of their attributes and operations are defined in a self-explanatory way. The RBAC security aspect definition also includes another class, RBACManager. This class is added to manipulate those four entities and the operations provided by this class can be used to crosscut other base UML operations. Two attributes of the "RBACManager" are two files: "RolePol-
icy" and "RightPolicy". These two attributes show that the RBAC aspect needs two files to describe the application’s business policies, i.e., role assignment policies and right assignment policies. Several aspect operations are defined to support the use of this aspect and they are referred as aspect operations:

- `distributeRole(U: User)`: distribute a role to a user;
- `updateRole(U: User)`: update a user’s role;
- `revokeRole(U: User)`: revoke a user’s role;
- `checkRole(U: User)`: check a user’s role;
- `createRole(F: File)`: create a new role from the role policy file;
- `deleteRole(R: Role)`: delete an existing role;
- `assignRight(R: Role)`: distribute a right/rights to a role;
- `updateRight(R: Role)`: update a role’s rights;
- `revokeRight(R: Role)`: revoke a role’s rights;
- `checkRight(R: Role)`: check a role’s right;
- `createRight(F: File)`: create a new right from the right policy file;
- `deleteRight(R: Right)`: delete an existing right;

Note that the MRO system needs to use aspect operations "createRole()", "createRight()", "distributeRole()" and "distributeRight()" to create roles and rights, then distribute roles to users and distribute rights to roles before it can use other aspect operations.
When a user logs into the MRO system and presents a access request to some object that is maintained by the system (for example, different documents), this request is forwarded to an access control object, which will check the role for this user using the RBAC security aspect operation "checkRole()". If the user has not any role associated with it, the access control object will distribute a role to this user according to the application’s access policy using RBAC security operation "distributeRole()". If the user has a role associated with it, the access control object will check the corresponding access rights that this role has using RBAC security aspect operation "checkRight()". If the user’s access request is in the set of access rights of the user’s role, the access request is permitted; otherwise, the access request is denied. In RBACAspect, the component RBACManager uses the classes User, Role, Right and Object. A base class of this aspect we consider here is AISDatabase defined in the section 8.2.1.2. The concrete users for this AISDatabase are Pratt & Whitney’ Employe and Costumer. The concrete roles for AISDatabase include Costumer, Forman, LineTech, Mechanic, Inspector, AHSEditor, TechnicalEditor, Administrator, Manager. Access rights include "View"(view data), "Modify" (modify data), "Delete" (delete data), "Debug" (debug AISDatabase system), "Report" (report a database problem). The objects are AISs. The detailed business policies, including the role assignment and right assignment, should be specified in the two files RolePolicy and RightPolicy defined in RBACManager. Table 8.VIII shows the weaving detail information for this aspect.
<table>
<thead>
<tr>
<th>Joinpoints</th>
<th>Pointcups</th>
<th>RBACAspect Operations</th>
<th>Crosscutting</th>
</tr>
</thead>
<tbody>
<tr>
<td>AISDatabase.initialize()</td>
<td>InitializeP(): execution(AISDatabase.initialize())</td>
<td>checkRole(), distributeRole(), checkRight()</td>
<td>Type: after</td>
</tr>
</tbody>
</table>

Table 8.VIII: Role-Based Access Control Aspect Weaving Information for AISDatabase.

### 8.2.1.4 Log for Audit Aspect

A security audit is a systematic evaluation of the security of a company’s information system by measuring how well it conforms to a set of established criteria. A thorough audit typically assesses the security of the system’s physical configuration and environment, software, information handling processes, and user practices. An important step in a security audit is the collection of records that describe a system’s activities, thus to help one understand the system’s behavior. Logging is one of the most common techniques that are used for this purpose. In its simplest form, logging prints messages describing the operations that the system performed. During the development cycle, logging plays a role similar to a debugger. It is also usually the only reasonable choice for debugging distributed programs. By examining the log, a developer can spot unexpected system behavior and subsequently correct it. A log also helps the developer see the interaction between different parts of a system in order to detect exactly where the problem might be [20].

The definition of this aspect which we name LogAspect, is based on the pattern defined in [20]. The main responsibility of the log for audit aspect is to collect useful data (i.e., what kinds of information should be logged), organize log data, and then provide an efficient way to analyze these data for the security audit process. Figure 8.6 shows LogAspect.
In the static view of the log for audit aspect, attributes include "loggingAPI", "logMsg" and "logFile":

- **initialization**: means that the operation initializeLoggingAPI() has to be used whenever the log for audit aspect is used. It is a boolean which initial value is false.

- **loggingAPI**: the logging API, which represents the underlying logging mechanism that this aspect is going to use, for example, the standard Java logging API and log4j from the Apache.

- **logMsg**: a string represents the message to be logged.

- **logFile**: the files that store the logs. The log for audit aspect may maintain one or multiple log files.

Operations of this aspect are:

- **initializeLoggingAPI(api: loggingAPI)**: initializes the logging application programming interface (i.e., parameter api), such as the standard Java logging API and log4j from the Apache.

- **exceptionLogging(msg: logMsg, file: logFile)**: logs exception throwing events in the system. Exception throwing is usually considered as an important event in the system, logging such occurrences is typically desirable. Exception logging
focuses on exceptional conditions in a program rather than execution of methods. Parameter msg is the message to be logged and may include the exception event type and the place where the exception happens, etc. Parameter file is the file that stores exception messages.

- **methodParameterLogging**(msg: logMsg, file: logFile): logs invoked objects and parameters of a method in the system. Parameter msg is the message to be logged and may include the name of the method, the names of invoked objects in this method, and the parameters of the method, etc. Parameter file is the file stores these messages.

- **methodCallTracing**(msg: logMsg, file: logFile): traces method calls in the system. Parameter msg is the message to be logged and may include the name of the method and the name of the class this method belongs to etc. Parameter file is the file that stores these messages.

- **regressionTesting**(msg: logMsg, file: logFile, inputs: string): regression testing is a process which tests a system to ensure that it still functions as expected as per specification when there is a change occurs to the system, for example, the change of the hardware platform, the addition of a new module or component, or a new version of the database, etc. Normally, in a regression testing process, a set of input that covers a sufficient range of possible options are provided and each time (i.e., when a change occurs) the output over the same input used to test the program/module are logged. This operation logs the output as well as the module’s intermediate steps. Then the actual log and the expected log are compared to check for the preservation of the system behavior. Parameter msg is the message to be logged and may include the name of the method being tested, and the name of the class this method belongs to, and the actual outputs of this method, etc. Parameter file is the file stores these messages. Parameter inputs is a string contains the inputs used to test the module and it should be customized as the suitable inputs for the real application.

- **timeProfiling**(msg: logMsg, file: logFile): collects the information about the time (in milliseconds or CPU cycle) spent by each method or procedure (i.e., basic blocks for sequence of instructions) in the system. Parameter msg is the message to be logged and may include the name of the method/procedure and the time of
the moment when the profiling happens in an appropriate format etc. Parameter
file is the file stores these messages.

- memoryProfiling(msg: logMsg, file: logFile): collects the information about the
  memory usage by each method or procedure (i.e., basic blocks for sequence of
  instructions) in the system. Parameter msg is the message to be logged and may
  include the name of the method/procedure and the memory usage of the moment
  when the profiling happens in an appropriate format etc. Parameter file is the file
  stores these messages.

- userActionLogging(msg: logMsg, file: logFile): in a situation where the interac-
  tion between users and the system is frequent, logging each user’s action is desir-
  able. This operation is used to log users’ actions towards the system. Parameter
  msg is the message to be logged and may include the information of a particular
  user (i.e., name, ID, etc.) and the user’s actions (i.e., requests, responses, etc.).
  Parameter file is the file stores these messages.

- recoveringLostData(file: logFile): recovers the lost data from log files.

- formatLogStatements(file: logFile): format the log messages in a log file, such
  as indents the log messages based on their depth in the transaction. This oper-
  ation can be independently used on a log file (or multiple files), or it can also
  be used by other operations in this aspect, namely, they are: exceptionLogging,
  methodParameterLogging, methodCallTracing, regressionTesting, timeProfiling,
  memoryProfiling, and userActionLogging.

- logAnalyzer(file: LogFile): processes the information contained in a log file, such
  as filtering data, extracting data, etc, thus to facilitate the security audit process.

- logFileSizeMonitor(files: LogFile[]): monitors the size of log files. Usually, a
  maximum size is specified for each log files. This operation monitors the size of
  log files, if there is any file reaches to the specified maximum file size, it will be
  moved to off-line storage.

- logLogic(): specifies logics that allow the suspension, resuming, and stop of other
  logging aspect operations: exceptionLogging, methodParameterLogging, method-
  CallTracing, regressionTesting, timeProfiling, memoryProfiling, and userAction-
Logging. This logLogic operation can be embedded in the body of these operations.

We have two constraints about this log aspect:

- The aspect operation "initializeLoggingAPI()" has to be used before any other aspect operations are used in the application.

- "timeProfiling()" and "memoryProfiling()", should crosscut the same UML operation twice at one time, both before and after the execution of the UML operation.

There are two invariants defined:

- The first invariant denotes the value of attribute "Initialization" should be true. The initial value of this attribute is "false" and its value can be changed to "true" in the aspect operation "initializeLoggingAPI()" (i.e., expressed as postcondition), which means that this operation has to be used whenever the log for audit aspect is used.

- The second invariant is defined in the context of "Package". The "Package" means the UML base model that the log for audit aspect woven into (i.e., normally, UML class diagrams). This constraint constraints that if there is any Crosscutting relationship involves aspect operation "timeProfiling()" or "memoryProfiling()", then its symmetric Crosscutting should be present too (i.e., "timeProfiling()" or "memoryProfiling()" should crosscut a joinpoint both "before" and "after").

LogAspect should crosscut almost all classes of the MRO application. However, for the sake of simplicity, we only use as base classes the AISDatabase and Mechanic classes. All operations of AISDatabase and Mechanic are joinpoints for LogAspect. But, also for the sake of simplicity, we will not use all of these operations as joinpoints. Here, we can choose to profile how much time it takes for the AISDatabase (server side) to process a request for a mechanic, i.e., when he execute its operation "chooseAIS()". Thus, the base class operation to consider in this context is "search()" of AISDatabase and the log for audit aspect operation selected is "timeProfiling()". Also, an important thing is the need to audit the user’s actions. Hence, we consider for example "assemble()" and "dis-assemble()" operations of Mechanic as joinpoints and the log for audit aspect operation selected is "userActionLogging()". Table 8.1X shows the weaving detail information for this aspect LogAspect.
<table>
<thead>
<tr>
<th>Joinpoints</th>
<th>Pointcuts</th>
<th>Log Audit Aspect Operations</th>
<th>Crosscutting</th>
</tr>
</thead>
<tbody>
<tr>
<td>AISDatabase.search()</td>
<td>SearchP(): execution(AISDatabase.search())</td>
<td>timeProfiling()</td>
<td>Type: before, Type: after</td>
</tr>
<tr>
<td>Mechanic.assemble(),</td>
<td>AssDisassembleP(): execution(Mechanic.assemble()) |</td>
<td>execution(Mechanic.disassemble())</td>
<td>userActionLogging()</td>
</tr>
<tr>
<td>Mechanic.disassemble()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.IX: Log for Audit Aspect Weaving Information for Mechanic and AISDatabase

### 8.2.1.5 File Versioning Aspect

Versioning is a technique for recording a history of changes to files. This history is useful for restoring previous versions of files, collecting a log of important changes over time, or to trace the file system activities of an intruder. Versioning is an important component of content creation and management. Version management is a key component of enterprise content management. Every modification to a file inherently results in a new version of the file. Instead of replacing the previous version with the new one, a versioning file system retains both. Users of such a system can then access any historical versions that the system keeps as well as the most recent one. Versioning file systems retain earlier versions of modified files, allowing recovery from user mistakes or system corruption. Applications of versioning include backups and disaster recovery, as well as monitoring intruders’ activities. File versioning is an important concern for Pratt & Whitney Canada’ employees (we notice this through a survey we conducted at Pratt & Whitney) because they usually may need to use a specific version of an AIS file at a given moment. They particularly asked to integrate this concern into MRO system. We name this file versioning aspect VersioningAspect. We represent this concern as aspect because it can tangle MRO system components such as AISDatabase. Figure 8.7 shows VersioningAspect.
Figure 8.7: Static View of Log for File Versioning Aspect.

In the static view of the log for audit aspect, attributes include:

- **versionNumber**: Every version has a unique version number. This number is used by the system to organize and keep track of the different versions of an AIS. The version number is automatically increased for each version that is created inside an AIS.

- **creationTime**: The creation time contains a timestamp pinpointing the exact date and time when the version was initially created. This information is set by the system and will remain the same regardless of what happens to the version.

- **modificationTime**: The modification time contains a timestamp revealing the exact date and time when the version was last modified. This information is set by the system every time the version is stored and when the version is finally published. When a version is published, the modification time of the object itself will be updated (it will simply be set to the same value as modification time of the version that was published).

- **creator**: The version’s creator contains a reference to the user that created the version. Although a content object can only belong to a single user (revealed by the "Owner" field), each version may belong different users. The creator reference is set by the system when the version is created. It cannot be manipulated and will not change even if the user who created the version is removed from the system.
status: The state of a version is determined by its status. There are two possibilities: a boolean value checkout indicating that a version is "checked-out" or not (see definition of check-out below), a boolean value "lock" indicating that a version is locked or not.

translation: The actual contents of a version is stored inside different translations. A translation is a representation of the information in a specific language. In other words, the translation layer allows a version of the object’s actual contents to exist in different languages. A version always has at least one translation of the content (which represents the data in the default/standard language).

Operations of this aspect are:

- change(vId1, vId2: int, f: file): A change (or diff, or delta) represents a specific modification to a document under version control. This method calculates the difference between two versions of a file.

- checkout(vId: int, f: file): A check-out (or co) is the act of creating a local working copy from the repository. A user may specify a specific revision or obtain the latest.

- commit(vId: int, f: file): A commit (checkin, ci, or, more rarely, install, submit or record) is the action of writing or merging the changes made in the working copy back to the repository.

- head(f: file): gets the most recent commit.

- merge(vId1, vId2: int, f: file): merges two versions of a file.

- lock(vId: int, f: file): lock a version of a file.

- unlock(vId: int, f: file): unlock a version of a file.

Table 8.X shows the weaving detail information for this aspect VersioningAspect. AISDatabase is a base class of this. We consider here two joinpoints: "AISDatabase.search()" and "AISDatabase.insert()". And the aspect operations we consider are "checkout()" and "commit()".
### Joinpoints

<table>
<thead>
<tr>
<th>Joinpoints</th>
<th>Pointcuts</th>
<th>Versioning Aspects Operations</th>
<th>Crosscutting</th>
</tr>
</thead>
<tbody>
<tr>
<td>AISDatabase.search()</td>
<td>SearchP(): execution(AISDatabase.search())</td>
<td>checkout()</td>
<td>Type: after</td>
</tr>
<tr>
<td>AISDatabase.insert()</td>
<td>InsertP(): execution(AISDatabase.insert())</td>
<td>commit()</td>
<td>Type: before</td>
</tr>
</tbody>
</table>

Table 8.X: File Versioning Aspect Weaving Information for AISDatabase

#### 8.2.2 Specification of the Case Study with the Modules

In this section, we specify the above components of the MRO system.

#### 8.2.2.1 Module Specification of the Class Mechanic

The module specification of the class Mechanic is \( MOD_M = \langle PAR_M, EXP_M, IMP_M, BOD_M \rangle \) with the four inclusion morphisms \( e_m, s_m, i_m, v_m \) such that: (figures 8.8, 8.9, 8.10, 8.11). The prescription axioms are pre, post

```plaintext
SPEC_CLASS PAR_M
  Sorts: string, ais, bool, int
END PAR_M

SPEC_CLASS IMP_M
  Sorts: string, ais, bool, int, saleOrder
  Actions:
    (one can put here actions of theses sorts)
END IMP_M
```

Figure 8.8: Module specification of Class Mechanic, Part 1

conditions constraints of the methods "assemble()" and "disassemble()".

#### 8.2.2.2 Module Specification of the Class AISDatabase

The module specification of the class AISDatabase is \( MOD_D = \langle PAR_D, EXP_D, IMP_D, BOD_D \rangle \) with the four inclusion morphisms \( e_d, s_d, i_d, v_d \) such that: (figures 8.12, 8.13, 8.14, 8.15).
\begin{figure}
\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{SPEC\_CLASS} $EXP_M$ \\
\hline
\textbf{Sorts:} \\
\hspace{1em} string, ais, bool, int, saleOrder, mechanic \\
\hline
\textbf{States:} \\
\hspace{1em} employedId: string $\rightarrow$ string \\
\hspace{1em} name: string $\rightarrow$ string \\
\hspace{1em} phone: string $\rightarrow$ string \\
\hspace{1em} email: string $\rightarrow$ string \\
\hline
\textbf{Actions:} \\
\hspace{1em} chooseAIS: int $\rightarrow$ ais \\
\hspace{1em} chooseSaleOrder: int $\rightarrow$ saleOrder \\
\hspace{1em} assemble: int int $\rightarrow$ \\
\hspace{1em} disassemble: int int $\rightarrow$ \\
\hspace{1em} postCheck: $\rightarrow$ bool \\
\hspace{1em} record: ais string int int $\rightarrow$ ais \\
\hline
\end{tabular}
\end{center}
\end{figure}

Figure 8.9: Module specification of Class Mechanic, Part 2

\begin{figure}
\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{SPEC\_CLASS} $BOD_M$ \\
\hline
\textbf{Sorts:} \\
\hspace{1em} string, ais, bool, int, saleOrder, mechanic \\
\hline
\textbf{States:} \\
\hspace{1em} employedId: string $\rightarrow$ string \\
\hspace{1em} name: string $\rightarrow$ string \\
\hspace{1em} phone: string $\rightarrow$ string \\
\hspace{1em} email: string $\rightarrow$ string \\
\hline
\textbf{Actions:} \\
\hspace{1em} chooseAIS: int $\rightarrow$ ais \\
\hspace{1em} chooseSaleOrder: int $\rightarrow$ saleOrder \\
\hspace{1em} assemble: int int $\rightarrow$ \\
\hspace{1em} disassemble: int int $\rightarrow$ \\
\hspace{1em} postCheck: $\rightarrow$ bool \\
\hspace{1em} record: ais string int int $\rightarrow$ ais \\
\hline
\textbf{Prescription Axioms:} \\
\hspace{1em} $G(\psi_a \rightarrow [\text{assemble()} \circ O(\psi_a)])$ \\
\hspace{1em} $G(\psi_d \rightarrow [\text{disassemble()} \circ O(\psi_d)])$ \\
\hline
\end{tabular}
\end{center}
\end{figure}

Figure 8.10: Module specification of Class Mechanic, Part 3

\begin{figure}
\begin{center}
\begin{tabular}{|l|}
\hline
$e_m: PAR_M \rightarrow EXP_M$ \\
$i_m: PAR_M \rightarrow IMP_M$ \\
$v_m: EXP_M \rightarrow BOD_M$ \\
$s_m: IMP_M \rightarrow BOD_M$ \\
\hline
\textbf{where} $v_m \circ e_m \circ s_m \circ i_m$ is satisfied. \\
\hline
\end{tabular}
\end{center}
\end{figure}

Figure 8.11: Module specification of Class Mechanic, Part 4
### SPEC_CLASS $PAR_D$
- **Sorts:** `string, ais, bool, int`
- **END $PAR_D$**

### SPEC_CLASS $IMP_D$
- **Sorts:** `string, ais, bool, int`
- **Actions:**
  (one can put here actions of theses sorts)
- **END $IMP_D$**

*Figure 8.12: Module specification of Class AISDatabase, Part 1*

### SPEC_CLASS $EXP_D$
- **Sorts:**
  `string, ais, bool, int, aisdatabase`
- **States:**
  `a : ais -> ais`
- **Actions:**
  - `delete : aisdatabase ais -> aisdatabase`
  - `search : string -> ais`
  - `update : aisdatabase string -> aisdatabase`
  - `insert : aisdatabase ais -> aisdatabase`
  - `initialize : -> aisdatabase`
- **END $EXP_D$**

*Figure 8.13: Module specification of Class AISDatabase, Part 2*

### SPEC_CLASS $BOD_D$
- **Sorts:**
  `string, ais, bool, aisdatabase`
- **States:**
  `a : ais -> ais`
- **Actions:**
  - `delete : aisdatabase ais -> aisdatabase`
  - `search : int string -> ais`
  - `update : aisdatabase string -> aisdatabase`
  - `insert : aisdatabase ais -> aisdatabase`
  - `initialize : -> aisdatabase`
- **END $BOD_D$**

*Figure 8.14: Module specification of Class AISDatabase, Part 3*
\[ e_d : \text{PAR}_D \rightarrow \text{EXP}_D \]
\[ i_d : \text{PAR}_D \rightarrow \text{IMP}_D \]
\[ v_d : \text{EXP}_D \rightarrow \text{BOD}_D \]
\[ s_d : \text{IMP}_D \rightarrow \text{BOD}_D \]

where \( v_d \circ e_d = s_d \circ i_d \) is satisfied.

Figure 8.15: Module specification of Class AISDatabase, Part 4

### 8.2.2.3 Module Specification of the Aspect AuthenticationAspect

The module specification of the aspect AuthenticationAspect is \( \text{MOD}_{\text{AUT}} = \langle \text{PAR}_{\text{AUT}}, \text{EXP}_{\text{AUT}}, \text{IMP}_{\text{AUT}}, \text{BOD}_{\text{AUT}} \rangle \) with the four inclusion morphisms \( e_{\text{aut}}, s_{\text{aut}}, i_{\text{aut}}, v_{\text{aut}} \) such that: (figures 8.16, 8.17, 8.18, 8.19).

<table>
<thead>
<tr>
<th>SPEC ASPECT \text{PAR}_{\text{AUT}}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sorts:</strong></td>
</tr>
<tr>
<td>string, bool, mechanic, ais, int, saleOrder</td>
</tr>
<tr>
<td><strong>END</strong> \text{PAR}_{\text{AUT}}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPEC ASPECT \text{IMP}_{\text{AUT}}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sorts:</strong></td>
</tr>
<tr>
<td>string, bool, mechanic, ais, int, saleOrder</td>
</tr>
<tr>
<td><strong>Joinpoints:</strong> \text{m} : mechanic, \text{s} : string</td>
</tr>
<tr>
<td>\text{forall} \text{m}, \text{execution}(\text{m.}*(..))</td>
</tr>
<tr>
<td><strong>Actions:</strong></td>
</tr>
<tr>
<td>(one can put here actions of theses sorts)</td>
</tr>
<tr>
<td><strong>END</strong> \text{IMP}_{\text{AUT}}</td>
</tr>
</tbody>
</table>

Figure 8.16: Module specification of Aspect AuthenticationAspect, Part 1
**Figure 8.17: Module specification of Aspect AuthenticationAspect, Part 2**

```
SPEC_ASPECT EXPAUT
Sorts:
  string, bool, mechanic, ais,
  int, saleOrder, authenticationAspect
Pointcut Axioms: m: mechanic
  \forall m, pr(execution(m.*(...)()))
Actions:
  getUser: string string
  verifyUser: \rightarrow bool
  getAccess:
  denyAccess:
  controlAccess:
END EXPAUT
```

**Figure 8.18: Module specification of Aspect AuthenticationAspect, Part 3**

```
SPEC_ASPECT BODAUT
Sorts:
  string, bool, mechanic, ais,
  int, saleOrder, authenticationAspect
Joinpoints: m: mechanic, s: string
  \forall m, execution(m.*(...)())
Pointcuts: m: mechanic, s: string
  \forall m, MROAccessP: execution(m.*(...)())
Actions:
  getUser: string string
  verifyUser: \rightarrow bool
  getAccess:
  denyAccess:
  controlAccess:
Pointcut Axioms: m: mechanic
  \forall m, pr(execution(m.*(...)()))
Advice Axioms: m: mechanic, au: authenticationAspect
  \forall m, (B_c.pr(MROAccessP))au.controlAccess()
END BODAUT
```

**Figure 8.19: Module specification of Aspect AuthenticationAspect, Part 4**

```
e_{aut} : PARAUT \rightarrow EXPAUT
i_{aut} : PARAUT \rightarrow IMPAUT
\nu_{aut} : EXPAUT \rightarrow BODAUT
s_{aut} : IMPAUT \rightarrow BODAUT
where \nu_{aut} \circ e_{aut} = s_{aut} \circ i_{aut} is satisfied.
```
8.2.2.4 Module Specification of the Aspect DOAspect

The module specification of the aspect DOAspect is $MOD_{DOA} = \langle PAR_{DOA}, EXP_{DOA}, IMP_{DOA}, BOD_{DOA} \rangle$ with the four inclusion morphisms $e_{doa}, s_{doa}, i_{doa}, v_{doa}$ such that: (figures 8.20, 8.21, 8.22, 8.24, 8.25, 8.23).

```
SPEC_ASPECT PAR_{DOA}
Sorts:
  string, bool, int, key, data,
  aisdatabase, mechanic, algorithm
END PAR_{DOA}
```

Figure 8.20: Module specification of Aspect DOAspect, Part 1

```
SPEC_ASPECT IMP_{DOA}
Sorts:
  string, bool, int, key, data,
  aisdatabase, mechanic, algorithm
Joinpoints: ad: aisdatabase, m: mechanic, s: string
  \forall ad, m, s, execution(ad.initialize()),
  execution(ad.search(s)), execution(m.read())
Actions:
  (one can put here actions of theses sorts)
END IMP_{DOA}
```

Figure 8.21: Module specification of Aspect DOAspect, Part 2
Figure 8.22: Module specification of Aspect DOAAspect, Part 3

Figure 8.23: Module specification of Aspect DOAAspect, Part 5
**SPEC_ASPECT** $BOD_{\text{DOA}}$

**Sorts:**
- string, bool, enum = \{RSA, DSA\}, key, data,
- aisdatabase, mechanic, algorithm, doaAspect

**Subsorts:**
- enum < string

**States:**
- algorithm: enum $\rightarrow$ enum
- decryption: bool $\rightarrow$ bool
- encryption: bool $\rightarrow$ bool
- priKey: string $\rightarrow$ string
- pubKey: string $\rightarrow$ string
- SIG: string $\rightarrow$ string

**Initial:**
- decryption = false, encryption = false

**Joinpoints:** ad: aisdatabase,
- $m$: mechanic, $s$: string

$\forall$ ad, $m$, $s$, execution(ad.initialize()),
execution(ad.search(s)), execution(m.read())

**Pointcuts:** ad: aisdatabase, $m$: mechanic, $s$: string

$\forall$ ad, $m$, $s$,
InitializeP: execution(ad.initialize())
SearchP: execution(ad.search(s))
ReadP: execution(m.read())

Figure 8.24: Module specification of Aspect DOAAspect, Part41
SPEC_ASPECT $BOD_{DOA}$

Actions:
- generateKey: int algorithm $\rightarrow$ key
- distributeKey: string key
- decryptData: key algorithm data
- encryptData: key algorithm data
- retrieveSIG: string $\rightarrow$ string

Pointcut Axioms: $ad: aisdatabase, m: mechanic, s: string$
$\forall ad, m, s, pr(execution(ad.initialize())), pr(execution(ad.search(s))), pr(execution(m.read()))$

Advice Axioms: $ad: aisdatabase, m: mechanic, s: string, d: doaAspect$
$\forall ad, m, s, d,$
$(A_f pr(InitializeP))d.generateKey(...)$
$(d.distributeKey(...); d.encryptData(...))$
$(A_f pr(SearchP))d.retrieveSIG(...)$
$(B_p pr(ReadP))d.decryptData(...)$

Prescription Axioms: $d: doaAspect$
$\forall d,$
$GO((d.encryption = d.decryption)?)$
$[encryptData()]0(encryption?)$
$[encryptData()]0(decryption?)$

END $BOD_{DOA}$

Figure 8.25: Module specification of Aspect DOAAspect, Part42

### 8.2.2.5 Module Specification of the Aspect RBACAspect

The module specification of the aspect RBACAspect is $MOD_{RBAC} = \langle PAR_{RBAC}, EXP_{RBAC}, IMP_{RBAC}, BOD_{RBAC} \rangle$ with the four inclusion morphisms $e_{rbac}, s_{rbac}, i_{rbac}, v_{rbac}$ such that: (figures 8.26, 8.27, 8.28, 8.29, 8.30, 8.31).

SPEC_ASPECT $PAR_{RBAC}$

Sorts:
- user, role, right, file,
- object, ais, aisdatabase, bool

END $PAR_{RBAC}$

Figure 8.26: Module specification of Aspect RBACAspect, Part1
SPEC_ASPECT $IMP_{RBAC}$

**Sorts:**
- user, role, right, file,
- object, ais, aisdatabase, bool

**Joinpoints:** ad: aisdatabase
- $\forall$ ad, execution(ad.initialize())

**Actions:**
- (one can put here actions of theses sorts)

END $IMP_{RBAC}$

Figure 8.27: Module specification of Aspect RBACAspect, Part2

SPEC_ASPECT $EXP_{RBAC}$

**Sorts:**
- user, role, right,
- object, ais, aisdatabase,
- rbacAspect, file, bool

**States:**
- rolePolicy: file $\rightarrow$ file
- right: file $\rightarrow$ file

**Pointcuts:** ad: aisdatabase
- $\forall$ ad
- InitializeP: execution(ad.initialize())

**Actions:**
- distributeRole: user $\rightarrow$ role
- updateRole: user $\rightarrow$ role
- revokeRole: user algorithm bool
- checkRole: user $\rightarrow$ role
- createRole: file $\rightarrow$ role
- deleteRole: role
- assignRight: role $\rightarrow$ right[]
- updateRight: role $\rightarrow$ right[]
- revokeRight: role $\rightarrow$ bool
- checkRight: role $\rightarrow$ right[]
- createRight: file $\rightarrow$ right
- deleteRight: right

END $EXP_{RBAC}$

Figure 8.28: Module specification of Aspect RBACAspect, Part3
SPEC_ASPECT $BOD_{RBAC}$

**Sorts:**
- user, role, right,
- object, ais, aisdatabase
- rbacAspect, file, bool

**States:**
- rolePolicy: file $\rightarrow$ file
- right: file $\rightarrow$ file

**Joinpoints:**
- ad: aisdatabase
  - $\forall$ ad, execution(ad.initialize())

**Pointcuts:**
- ad: aisdatabase
  - $\forall$ ad,
    - InitializeP: execution(ad.initialize())

**Actions:**
- distributeRole: user $\rightarrow$ role
- updateRole: user $\rightarrow$ role
- revokeRole: user algorithm bool
- checkRole: user $\rightarrow$ role
- createRole: file $\rightarrow$ role
- deleteRole: role
- assignRight: role $\rightarrow$ right[
- updateRight: role $\rightarrow$ right[
- revokeRight: role $\rightarrow$ bool
- checkRight: role $\rightarrow$ right[
- createRight: file $\rightarrow$ right
- deleteRight: right

Figure 8.29: Module specification of Aspect RBACAspect, Part41
SPEC_ASPECT $BOD_{RBAC}$

Pointcut Axioms: ad: aisdatabase
\[ \forall \text{ad, } pr(\text{execution(ad.initialize()))} \]

Advice Axioms: ad: aisdatabase, rb: rbacAspect
\[ \forall \text{ad, } \text{rb}, \]
\[ (A/pr(\text{InitializeP}))\text{rb.checkRole}(...) \]
\[ ;(\varphi_r \rightarrow \text{rb.distributeRole}(...)\text{/checkRight}) \]

Prescription Axioms: rb: rbacAspect
\[ \forall \text{rb}, \]
\[ \text{I(rb.createRight;rb.createRole)} \]
\[ \text{I(rb.distributeRole;rb.createRight)} \]
\[ \text{I(rb.distributeRight;rb.distributeRole)} \]
\[ \forall \text{a } \in \{\text{updateRole, revokeRole, deleteRole, updateRight, revokeRight, checkRight, deleteRight}\} \]

END $BOD_{RBAC}$

Figure 8.30: Module specification of Aspect RBACAspect, Part 42

\[
\begin{align*}
\epsilon_{rbac} : PAR_{RBAC} & \rightarrow EXP_{RBAC} \\
\iota_{rbac} : PAR_{RBAC} & \rightarrow IMP_{RBAC} \\
v_{rbac} : EXP_{RBAC} & \rightarrow BOD_{RBAC} \\
s_{rbac} : IMP_{RBAC} & \rightarrow BOD_{RBAC}
\end{align*}
\]

where $v_{rbac} \circ \epsilon_{rbac} = s_{rbac} \circ \iota_{rbac}$ is satisfied.

Figure 8.31: Module specification of Aspect RBACAspect, Part 5

$\varphi_r$ is the proposition "user has this role". The prescription axioms of the body part means that the MRO system needs to use aspect operations "createRole()", "createRight()", "distributeRole()" and "distributeRight()" to create roles and rights, then distribute roles to users and distribute rights to roles before it can use other aspect operations.

8.2.2.6 Module Specification of the Aspect LogAspect

The module specification of the aspect RBACAspect is $MOD_{Log} = \langle PAR_{Log}, EXP_{Log}, IMP_{Log}, BOD_{Log} \rangle$ with the four inclusion morphisms $\epsilon_{log}, s_{log}, i_{log}, v_{log}$ such that: (figures 8.32, 8.33, 8.34, 8.35, 8.36).
<table>
<thead>
<tr>
<th>SPEC_ASPECT</th>
<th>PAR_{Log}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sorts:</strong></td>
<td>loggingAPI, file, logFile, string, bool, logMsg, aisdatabase, mechanic</td>
</tr>
<tr>
<td><strong>END</strong></td>
<td>PAR_{Log}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPEC_ASPECT</th>
<th>IMP_{Log}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sorts:</strong></td>
<td>loggingAPI, file, logFile, string, bool, logMsg, aisdatabase, mechanic</td>
</tr>
<tr>
<td><strong>Subsorts:</strong></td>
<td>logFile &lt; file, logMsg &lt; string</td>
</tr>
<tr>
<td><strong>Joinpoints:</strong></td>
<td>ad: aisdatabase, m: mechanic</td>
</tr>
<tr>
<td></td>
<td>$\forall$ ad, execution(ad.search())</td>
</tr>
<tr>
<td></td>
<td>$\forall$ m, execution(m.assemble())</td>
</tr>
<tr>
<td><strong>Actions:</strong></td>
<td>(one can put here actions of theses sorts)</td>
</tr>
<tr>
<td><strong>END</strong></td>
<td>IMP_{Log}</td>
</tr>
</tbody>
</table>

Figure 8.32: Module specification of Aspect LogAspect, Part1
SPEC_ASPECT $EXP_{Log}$

Sorts:
- loggingAPI, file, logFile, string, bool,
- logMsg, aisdatabase, mechanic, logAspect

Subsorts:
- logFile < file, logMsg < string

States:
- initialization: bool $\rightarrow$ bool
- loggingAPI: loggingAPI $\rightarrow$ loggingAPI
- logMsg: logMsg $\rightarrow$ logMsg
- logFile: logFile $\rightarrow$ logFile

Pointcuts:
- $\forall$ ad, m, SearchP: execution(ad.search())
  AssDisassembleP: execution(m.assemble()) || execution(m.disassemble())

Actions:
- initializeLoggingAPI: logginAPI
- exceptionLogging: logMSG logFile
- methodParameterLogging: logMSG logFile
- methodCallTracing: logMSG logFile
- regressionTesting: logMSG logFile string
- timeProfiling: logMSG logFile
- memoryProfiling: logMSG logFile
- userActionLogging: logMSG logFile
- recoveringLostData: logFile
- formatLogStatements: logFile
- logAnalyzer: logFile
- logFileSizeMonitor: logFile[]
- logLogic:

END $EXP_{Log}$

Figure 8.33: Module specification of Aspect LogAspect, Part2
SPEC_ASPECT $BOD_{Log}$

Sorts:
loggingAPI, file, logFile, string, bool,
logMsg, aisdatabase, mechanic, logAspect

Subsorts:
logFile < file, logMsg < string

States:
initialization: bool $\rightarrow$ bool
loggingAPI: loggingAPI $\rightarrow$ loggingAPI
logMsg: logMsg $\rightarrow$ logMsg
logFile: logFile $\rightarrow$ logFile

Initial:
initialization = false

Joinpoints:
ad: aisdatabase, $m$: mechanic
$\forall$ ad, execution(ad.search())
$\forall$ $m$, execution(m.assemble()) $\mid\mid$
execution(m.disassemble())

Pointcuts:
ad: aisdatabase, $m$: mechanic
$\forall$ ad, $m$, SearchP: execution(ad.search())
AssDisassembleP: execution(m.assemble()) $\mid\mid$
execution(m.disassemble())

Actions:
initializeLoggingAPI: logginAPI
exceptionLogging: logMSG logFile
methodParameterLogging: logMSG logFile
methodCallTracing: logMSG logFile
regressionTesting: logMSG logFile string
timeProfiling: logMSG logFile
memoryProfiling: logMSG logFile
userActionLogging: logMSG logFile
recoveringLostData: logFile
formatLogStatements: logFile
logAnalyzer: logFile
logFileSizeMonitor: logFile[]
logLogic:
SPEC_ASPECT $BOD_{\text{Log}}$

Pointcut Axioms: \( ad: \) aisdatabase, \( m: \) mechanic
\[
\forall \text{ad}, m, \text{pr}(\text{execution(ad.search())}), \\
\text{pr}(\text{execution(m.assemble())}) \\
|| \text{execution(m.disassemble())}
\]

Advice Axioms: \( ad: \) aisdatabase, \( m: \) mechanic,
\[
\forall \text{ad}, \text{lg}, m, \\
(A, \text{pr(SearchP)}) \text{lg}.\text{timeProfiling}(\ldots) \text{lo}.\text{timeProfiling}(\ldots) \\
(A, \text{pr(AssDisassembleP)}) \text{lg}.\text{userActionLogging}(\ldots)
\]

Prescription Axioms: \( \text{lg}: \) logAspect
\[
\forall \text{lg}, \\
\forall \phi \in \{\text{timeProfiling()}, \text{memoryProfiling}()\}, \\
(A_{/}\phi)\beta \rightarrow O((B_{/}\phi)\beta)\?, \\
(B_{/}\phi)\beta \rightarrow O((A_{/}\phi)\beta)\?
\]

[initializeLoggingAPI()]\text{initialization}

END $BOD_{\text{Log}}$

Figure 8.35: Module specification of Aspect LogAspect, Part32

\[
\begin{align*}
\epsilon_{\text{log}} &: PAR_{\text{Log}} \rightarrow EXP_{\text{Log}} \\
i_{\text{log}} &: PAR_{\text{Log}} \rightarrow IMP_{\text{Log}} \\
v_{\text{log}} &: EXP_{\text{Log}} \rightarrow BOD_{\text{Log}} \\
s_{\text{log}} &: IMP_{\text{Log}} \rightarrow BOD_{\text{Log}}
\end{align*}
\]

where \( v_{\text{log}} \circ \epsilon_{\text{log}} = s_{\text{log}} \circ i_{\text{log}} \) is satisfied.

Figure 8.36: Module specification of Aspect LogAspect, Part 4

The prescription axioms represent the above constraints and invariants. The second invariant and the second constraint are specified by the same formulae. \( \phi \) is related to any joinpoint of any base class of this aspect.

8.2.2.7 Module Specification of the Aspect VersioningAspect

The module specification of the aspect VersioningAspect is \( MOD_{\text{Ver}} = (PAR_{\text{Ver}}, EXP_{\text{Ver}}, IMP_{\text{Ver}}, BOD_{\text{Ver}}) \) with the four inclusion morphisms
ever, sver, iVer, vver such that: (figures 8.37, 8.38, 8.39, 8.40, 8.41, 8.42).

```
SPEC_ASPECT PARVer
Sorts:
  user, file, string,
  bool, aisdatabase, int
END PARVer
```

Figure 8.37: Module specification of Aspect VersionningAspect, Part1

```
SPEC_ASPECT IMPVer
Sorts:
  user, file, string,
  bool, aisdatabase, int
Joinpoints: ad: aisdatabase
  ∀ ad, execution(ad.search())
    || execution(ad.insert())
Actions:
  (one can put here actions of theses sorts)
END IMPVer
```

Figure 8.38: Module specification of Aspect VersionningAspect, Part2
SPEC_ASPECT \texttt{EXP}_{\text{Ver}}

\textbf{Sorts:}

versioningAspect, file, string, bool, aisdatabase, int, user
enum = \{checkout, locked\}

\textbf{States:}

versionNumber: int \rightarrow int
creationTime: string \rightarrow string
modificationTime: string \rightarrow string
creator: user \rightarrow user
status: enum \rightarrow enum
translation: string \rightarrow string

\textbf{Pointcuts:}

ad: aisdatabase
\forall ad,
SearchP: execution(ad.search())
InsertP: execution(ad.insert())

\textbf{Actions:}

change: int int file \rightarrow file
checkout: int file \rightarrow file
commit: int file
head: \rightarrow file
merge: int int file \rightarrow file
lock: int file \rightarrow bool
unlock: int file \rightarrow bool

\text{END \texttt{EXP}_{\text{Ver}}}

Figure 8.39: Module specification of Aspect VersionningAspect, Part 3
**SPEC_ASPECT** \( BOD\text{\textsubscript{Ver}} \)

**Sorts:**
- versionningAspect, file, string, bool, aisdatabase, int, user
- enum = \{checkout, locked\}

**States:**
- versionNumber: int \( \rightarrow \) int
- creationTime: string \( \rightarrow \) string
- modificationTime: string \( \rightarrow \) string
- creator: user \( \rightarrow \) user
- status: enum \( \rightarrow \) enum
- translation: string \( \rightarrow \) string

**Joinpoints:** ad: aisdatabase
- \( \forall \) ad, execution(ad.search())
  - || execution(ad.insert())

**Pointcuts:** ad: aisdatabase
- \( \forall \) ad,
  - SearchP: execution(ad.search())
  - InsertP: execution(ad.insert())

**Actions:**
- change: int int file \( \rightarrow \) file
- checkout: int file \( \rightarrow \) file
- commit: int file
- head: \( \rightarrow \) file
- merge: int int file \( \rightarrow \) file
- lock: int file \( \rightarrow \) bool
- unlock: int file \( \rightarrow \) bool

Figure 8.40: Module specification of Aspect VersionningAspect, Part41


\[ SPEC\_ASPECT\ BOD_{\text{Ver}} \]

**Pointcut Axioms:** ad: aisdatabase
\[ \forall \text{ad}, \text{pr}(\text{execution}(\text{ad}.\text{search}())), \text{pr}(\text{execution}(\text{ad}.\text{insert}())) \]

**Advice Axioms:** ad: aisdatabase, ve: versionningAspect
\[ \forall \text{ad}, \text{ve}, \]
\[ (A_{\text{pr}}(\text{SearchP}))\text{ve}.\text{checkout(...)} \]
\[ (B_{\text{pr}}(\text{InsertP}))\text{ve}.\text{commit(...)} \]

END \ BOD_{\text{Ver}}

Figure 8.41: Module specification of Aspect VersionningAspect, Part 42

\[ \epsilon_{\text{ver}} : PAR_{\text{ver}} \rightarrow EXP_{\text{ver}} \]
\[ i_{\text{ver}} : PAR_{\text{ver}} \rightarrow IMP_{\text{ver}} \]
\[ v_{\text{ver}} : EXP_{\text{ver}} \rightarrow BOD_{\text{ver}} \]
\[ s_{\text{ver}} : IMP_{\text{ver}} \rightarrow BOD_{\text{ver}} \]

where \( v_{\text{ver}} \circ \epsilon_{\text{ver}} = s_{\text{ver}} \circ i_{\text{ver}} \) is satisfied.

Figure 8.42: Module specification of Aspect VersionningAspect, Part 5

### 8.2.2.8 Module Specification of the Aspect CoordinationationAspect

The coordination module aspect will mainly contain the societal life properties under the field Prescription Axioms. It records information about all aspect modules which are present in the application and also the classes. This aspect will contain the prevention mechanism properties described in chapter 7. This coordination aspect module is responsible for the management of the aspect scheduling or ordering at the shared join points. This coordination aspect reinforces the concept of the “Separation of Concerns”.

We have aspect precedence constraints in this application. AuthenticationAspect should be executed before any other aspect in the system, because of the security policy. This aspect precedence constraint is formalized as follows: let au: authenticationAspect, do: doaAspect, lo: LogAspect,

\[ (B_{\text{epr}}(\text{ReadP}()))(I(\text{do.decryptData}; \text{au.controlAccess})) \]
\[ (B_{\text{epr}}(\text{AssDisassembleP}()))(I(\text{lo.userActionLogging}; \text{au.controlAccess})) \lor \]
\[ (A_{\text{fpr}}(\text{AssDisassembleP}()))(I(\text{lo.userActionLogging}; \text{au.controlAccess})) \]
Semantically, RBACAspect should be executed before DOAAstpect. This aspect precedence constraint is formalized as follows: let rb: RBACAspect, do: doaAspect,

\[(A_fpr(InitializeP()))(I(do.\alpha; rb.\beta))\]
\[\forall \alpha \in \{\text{generateKey, distributeKey, encryptData}\} \text{ and } \forall \beta \in \{\text{checkRole, distributeRole, checkRight}\}\]

Also, DOAAstpect should be executed before LogAspect because LogAspect will log operation behaviors after these operations have been performed. This aspect precedence constraint is formalized as follows: let do: doaAspect, lo: LogAspect,

\[(A_fpr(SearchP()))(I(lo.timeProfiling; do.retrieveSIG))\]

We have also to prevent pointcut fault types by the following formulae:

\[\forall m, (B_fpr(MROAccessP))au.controlAccess()\]

\[\forall ad, m, s, d, (A_fpr(InitializeP))d.generateKey(...)d.distributeKey(...)d.encryptData(...)\]
\[(A_fpr(SearchP))d.retrievalSIG(...)\]
\[(B_fpr(ReadP))d.decryptData(...)\]

\[\forall ad, rb, (A_fpr(InitializeP))rb.checkRole(...); (\phi_r \rightarrow rb.distributeRole(...)/checkRight)\]

\[\forall ad, lg, m, (B_fpr(SearchP))lg.timeProfiling(...)\]
\[(A_fpr(SearchP))lg.timeProfiling(...)\]
\[(A_fpr(AssembleDisassembleP))lg.userActionLogging(...)\]

\[\forall ad, ve, (A_fpr(SearchP))ve.checkout(...)\]
\[(B_fpr(InsertP))ve.commit(...)\]

All of these constraints will be put under the prescription axiom’s field of the body part of CoordinationAspect.

The module specification of the aspect CoordinationAspect is \(MOD_{Coo} = (PAR_{Coo}, EXP_{Coo}, IMP_{Coo}, BOD_{Coo})\) with the four inclusion morphisms \(e_{Coo}, s_{Coo}, i_{Coo}, v_{Coo}\) such that \(EXP_{Coo} = IMP_{Coo}\) and \(PAR_{Coo}, IMP_{Coo}, BOD_{Coo}\) are shown in the figures 8.43, 8.44, 8.45, 8.46.
**SPEC_ASPECT** \( PAR_{C00} \)

**Sorts:**
- mechanic, aisdatabase, authenticationAspect, doaAspect
- rbacAspect, logAspect, versionningAspect

**END** \( PAR_{C00} \)

Figure 8.43: Module specification of Aspect CoordinationAspect, Part 1

**SPEC_ASPECT** \( IMP_{C00} \)

**Sorts:**
- mechanic, aisdatabase, authenticationAspect, doaAspect
- rbacAspect, logAspect, versionningAspect

**Joinpoints:** \( ad: \) aisdatabase, \( m: \) mechanic, \( s: \) string
- \( \forall m, \) execution(\( m.*(...) \))
- \( \forall ad, s, \) execution(\( ad.search(s) \))
- \( || \) execution(\( ad.insert() \)) \( || \) execution(\( ad.initialize() \))

**Actions:**
- (one can put here actions of these sorts)

**END** \( IMP_{C00} \)

Figure 8.44: Module specification of Aspect CoordinationAspect, Part 2
SPEC_ASPECT $BOD_{\text{Coo}}$

Sorts:
- mechanic, aisdatabase, authenticationAspect, doaAspect
- rbacAspect, logAspect, versionningAspect, coordinationAspect

Prescription Axioms:
- $au$: authenticationAspect, $do$: doaAspect, $ad$: aisdatabase,
- $ve$: versionningAspect
- $lo$: LogAspect, $rb$: RBACAspect,
- $m$: mechanic
- $\forall au, do, lo, rb$, $(B_{pr}(\text{ReadP}())) \cdot (I(\text{do.decryptData}; au.\text{controlAccess}))$
- $(B_{pr}(\text{AssDisassembleP}())) \cdot (I(\text{lo.userActionLogging}; au.\text{controlAccess} \lor$
- $(A_{pr}(\text{AssDisassembleP}())) \cdot (I(\text{lo.userActionLogging}; au.\text{controlAccess}))$
- $(A_{pr}(\text{InitializeP}())) \cdot (I(\text{do.} \alpha; rb. \beta))$
- $\forall \alpha \in \{\text{generateKey, distributeKey, encryptData}\}$ and
- $\forall \beta \in \{\text{checkRole, distributeRole, checkRight}\}$
- $(A_{pr}(\text{SearchP}())) \cdot (I(\text{lo.timeProfiling}; do.\text{retrieveSIG}))$
- $\forall m, (B_{pr}(\text{MROAccessP})); au.\text{controlAccess}()$
- $\forall ad, m, s, do$,
- $(A_{pr}(\text{InitializeP})); do.\text{generateKey}(...)$
- $; do.\text{distributeKey}(...); do.\text{encryptData}(...)$
- $(A_{pr}(\text{SearchP})); do.\text{retrieveSIG}(...)$
- $(B_{pr}(\text{ReadP})); do.\text{decryptData}(...)$
- $\forall ad, rb$,
- $(A_{pr}(\text{InitializeP})); rb.\text{checkRole}(...)$
- $(\phi, \rightarrow rb.\text{distributeRole}(...)/\text{checkRight})$
- $\forall ad, lo, m$,
- $(B_{pr}(\text{SearchP})); lo.\text{timeProfiling}(...)$
- $(A_{pr}(\text{SearchP})); lo.\text{timeProfiling}(...)$
- $(A_{pr}(\text{AssDisassembleP})); lo.\text{userActionLogging}(...)$
- $\forall ad, ve$,
- $(A_{pr}(\text{SearchP})); ve.\text{checkout}(...)$
- $(B_{pr}(\text{InsertP})); ve.\text{commit}(...)$

END $BOD_{\text{Coo}}$

Figure 8.45: Module specification of Aspect CoordinationAspect, Part3
8.3 Weaving

As stated in chapter 7, section 7.4, before weaving any other aspect, we first weave the coordination aspect into the base modules. By weaving the coordination aspect into the base modules, all prescription axioms of this coordination Aspect become theorems of the modular system. For the sake of simplicity, we will just consider the base module "Mechanic" and only one aspect more than "CoordinationAspect". The weaving process is same of all other base modules and aspects.

8.3.1 Weaving of CoordinationAspect into the Mechanic Class

Let $MOD_{S1} = \langle PAR_{S1}, EXP_{S1}, IMP_{S1}, BOD_{S1} \rangle$ be the connector module of $MOD_M$ and $MOD_{Coo}$, with the four inclusion morphisms $e_{s1}, s_{s1}, i_{s1}, v_{s1}$ such that:

$PAR_{S1} = PAR_M \cap PAR_{Coo} = \emptyset$

$EXP_{S1} = EXP_M \cap EXP_{Coo}$

$IMP_{S1} = IMP_M \cap IMP_{Coo} = \emptyset$

$BOD_{S1} = (BOD_M \cap BOD_{Coo} \cap \{jp \in Ad.TJ, \forall Ad \in MOD_{Coo}.TAD\}) \cup \{shared\ sorts\} = \emptyset$, because there is no advice in $MOD_{Coo}$.

The rest of the specification of $MOD_{S1}$ is represented in the figures 8.47, 8.48.

```plaintext
SPEC_ASPECT EXP_{S1}
Sorts:
mechanic
END EXP_{S1}
```

Figure 8.47: Module specification of $MOD_{S1}$, Part1

$\epsilon_{Coo} : PAR_{Coo} \rightarrow EXP_{Coo}$

$i_{Coo} : PAR_{Coo} \rightarrow IMP_{Coo}$

$v_{Coo} : EXP_{Coo} \rightarrow BOD_{Coo}$

$s_{Coo} : IMP_{Coo} \rightarrow BOD_{Coo}$

where $v_{Coo} o e_{Coo} = s_{Coo} o i_{Coo}$ is satisfied.
Figure 8.48: Module specification of \( \text{MOD}_{S1} \), Part 2

The two module specification morphisms \( f_1 : \text{MOD}_{S1} \rightarrow \text{MOD}_{\text{COO}} \) and \( g_1 : \text{MOD}_{S1} \rightarrow \text{MOD}_{M} \) are inclusion morphisms. It is worth to notice that a join-point of \( \text{MOD}_{S2} \) can be related to an element of \( \text{MOD}_{M} \), no matter of the field of \( \text{MOD}_{M} \) this later belongs to. By applying the weaving algorithm 1 of chapter 6, we first compute \( M_{w1} \) (union of \( \text{MOD}_{M} \) and \( \text{MOD}_{\text{COO}} \)) and then \( \text{MOD}_{w1} \).

\( \text{MOD}_{w1} = \text{MOD}_{u1} \) (in which we possibly remove certain elements) because there is no advice in \( \text{MOD}_{\text{COO}} \). \( \text{MOD}_{w1} \) is the result of the weaving of \( \text{MOD}_{M} \) with \( \text{MOD}_{\text{COO}} \).

\( \text{MOD}_{w1} = \langle \text{PAR}_{w1}, \text{EXP}_{w1}, \text{IMP}_{w1}, \text{BOD}_{w1} \rangle \) is defined by the figures 8.49, 8.50, 8.51, 8.52, and 8.53.

Figure 8.49: Module specification of the weaving of Mechanic with CoordinationAspect, Part 1
\textbf{SPEC\_CLASS} \textit{EXP}_{w_1} \\
\textbf{Sorts:} \\
string, ais, bool, int, saleOrder, mechanic, ais\_database, authenticationAspect, doaAspect, rbacAspect, logAspect, versionningAspect, coordinationAspect \\
\textbf{States:} \\
employedId: string $\rightarrow$ string \\
name: string $\rightarrow$ string \\
phone: string $\rightarrow$ string \\
email: string $\rightarrow$ string \\
\textbf{Actions:} \\
chooseAIS: int $\rightarrow$ ais \\
chooseSaleOrder: int $\rightarrow$ saleOrder \\
assemble: int int \\
disassemble: int int $\rightarrow$ \\
postCheck: $\rightarrow$ bool $\rightarrow$ \\
record: ais string int int $\rightarrow$ ais \\
\textbf{END} \textit{EXP}_{w_1}

Figure 8.50: Module specification of the weaving of Mechanic with CoordinationAspect, Part2
**SPEC_CLASS** \( BOD_{w1} \)

**Sorts:**
- string, ais, bool, int, saleOrder,
- mechanic, aisdatabase, authenticationAspect, doaAspect,
- rbacAspect, logAspect, versionningAspect, coordinationAspect

**States:**
- employedId: string \( \rightarrow \) string
- name: string \( \rightarrow \) string
- phone: string \( \rightarrow \) string
- email: string \( \rightarrow \) string

**Actions:**
- chooseAIS: int \( \rightarrow \) ais
- chooseSaleOrder: int \( \rightarrow \) saleOrder
- assemble: int int
- disassemble: int int
- postCheck: \( \rightarrow \) bool
- record: ais string int int \( \rightarrow \) ais

Figure 8.51: Module specification of the weaving of Mechanic with CoordinationAspect, Part3
SPEC_CLASS $BOD_{w1}$

**Prescription Axioms:**

- $au$: authenticationAspect, $do$: doaAspect, $ad$: aisdatabase, $ve$: versionningAspect
- $lo$: LogAspect, $rb$: RBACAspect, $m$: mechanic
- $G(q_u \rightarrow \text{assemble()}O(\psi_u))$
- $G(q_d \rightarrow \text{disassemble()}O(\psi_d))$
- $\forall au, do, lo, rb,$
  - $(B_{pr}(\text{ReadP}()))) (I(do.decryptData; au.controlAccess))$
  - $(B_{pr}(\text{AssDisassembleP}()))) (I(lo.userActionLogging; au.controlAccess \lor$
  - $(A_{pr}(\text{AssDisassembleP}()))) (I(lo.userActionLogging; au.controlAccess))$
  - $(A_{pr}(\text{InitializeP}()))) (I(do.a; rb.\beta)$
  - $\forall \alpha \in \{\text{generateKey, distributeKey, encryptData}\}$ and
  - $\forall \beta \in \{\text{checkRole, distributeRole, checkRight}\}$
  - $(A_{pr}(\text{SearchP}()))) (I(lo.timeProfiling; do.retrieveSIG))$
- $\forall m, (B_{pr}(\text{MROAccessP})) au.controlAccess()$
- $\forall ad, m, s, do,$
  - $(A_{pr}(\text{InitializeP})) do.\text{generateKey}(...)$
  - $; do.\text{distributeKey}(...); do.\text{encryptData}(...)$
  - $(A_{pr}(\text{SearchP})) do.\text{retrieveSIG}(...)$
  - $(B_{pr}(\text{ReadP})) do.\text{decryptData}(...)$
- $\forall ad, rb, (A_{pr}(\text{InitializeP})) rb.\text{checkRole}(...)$
  - $; rb.\text{distributeRole}(...)/\text{checkRight}$
- $\forall ad, lo, m, (B_{pr}(\text{SearchP})) lo.\text{timeProfiling}(...)$
  - $(A_{pr}(\text{SearchP})) lo.\text{timeProfiling}(...)$
  - $(A_{pr}(\text{AssDisassembleP})) lo.\text{userActionLogging}(...)$
- $\forall ad, ve, (A_{pr}(\text{SearchP})) ve.\text{checkout}(...)$
  - $(B_{pr}(\text{InsertP})) ve.\text{commit}(...)$

**END $BOD_{w1}$**

Figure 8.52: Module specification of the weaving of Mechanic with CoordinationAspect, Part4
8.3.2 Weaving of AuthenticationAspect into \( MOD_{w1} \)

Let \( MOD_{S2} = \langle PAR_{S2}, EXP_{S2}, IMP_{S2}, BOD_{S2} \rangle \) be the connector module of \( MOD_{w1} \) and \( MOD_{AUT} \) with the four inclusion morphisms \( e_{s2}, s_{s2}, i_{s2}, v_{s2} \) such that:

\[
\begin{align*}
PAR_{S2} &= PAR_{w1} \cap PAR_{AUT} \\
EXP_{S2} &= EXP_{w1} \cap EXP_{AUT} \\
IMP_{S2} &= IMP_{w1} \cap IMP_{AUT} \\
BOD_{S2} &= (BOD_{w1} \cap BOD_{AUT} \cap \{jp \in Ad.TJ, \forall Ad \in MOD_{AUT\cdot TAD}\}) \cup \{\text{shared sorts}\}.
\end{align*}
\]

\( MOD_{S2} \) is represented in the figures 8.54, 8.55 and 8.56.

---

```
SPEC_CLASS PAR_{S2}
Sorts: string, bool, int, mechanic
END PAR_{S2}
```

```
SPEC_CLASS IMP_{S2}
Sorts: string, bool, mechanic, int, saleOrder
Actions: (classical operations on these sorts)
END IMP_{S2}
```

Figure 8.54: Module specification of \( MOD_{S2} \), Part 1
**SPEC_CLASS** $EXP_{S2}$

**Sorts:**
- string, bool, mechanic, ais,
- int, saleOrder, authenticationAspect

**END** $EXP_{S2}$

Figure 8.55: Module specification of $MOD_{S2}$, Part 2

**SPEC_CLASS** $BOD_{S2}$

**Sorts:**
- string, bool, mechanic, ais,
- int, saleOrder, authenticationAspect

**Joinpoints:** $m$: mechanic
- $\forall m$, execution$(m.*(..))$

**End** $BOD_{S2}$

| $e_{S2}$ : $PAR_{S2} \rightarrow EXP_{S2}$ |
| $i_{S2}$ : $PAR_{S2} \rightarrow IMP_{S2}$ |
| $v_{S2}$ : $EXP_{S2} \rightarrow BOD_{S2}$ |
| $s_{S2}$ : $IMP_{S2} \rightarrow BOD_{S2}$ |

where $v_{S2} \circ e_{S2} = s_{S2} \circ i_{S2}$ is satisfied.

Figure 8.56: Module specification of $MOD_{S2}$, Part 3

The two module specification morphisms $f_{2} : MOD_{S2} \rightarrow MOD_{AUT}$ and $g_{2} : MOD_{S2} \rightarrow MOD_{w1}$ are inclusion morphisms. It is worth to notice that a joinpoint of $MOD_{S2}$ can be related to an element of $MOD_{w1}$, no matter of the field of $MOD_{w1}$ this later belongs to.

By applying the weaving algorithm 1 (chapter 6), we first compute $M_{u2}$ (union of $MOD_{w1}$ and $MOD_{AUT}$) and then $MOD_{w2}$. $MOD_{w2}$ is the result of the weaving of $MOD_{w1}$ with $MOD_{AUT}$. $MOD_{u2} = \langle PAR_{u2}, EXP_{u2}, IMP_{u2}, BOD_{u2} \rangle$ is defined by the figures 8.58, 8.59, 8.60, 8.61, and 8.62.
Figure 8.57: Module specification of the $MOD_{a2}$, Part1

Figure 8.58: Module specification of the $MOD_{a2}$, Part1
**SPEC_ASPECT** $EXP_{u2}$

**Sorts:**
- string, ais, bool, int, saleOrder,
- mechanic, aisdatabase, authenticationAspect, doaAspect,
- rbacAspect, logAspect, versionningAspect, coordinationAspect

**States:**
- employedId: string $\rightarrow$ string
- name: string $\rightarrow$ string
- phone: string $\rightarrow$ string
- email: string $\rightarrow$ string

**Pointcut Axioms:** $m$: mechanic
\[ \forall m, \text{pr}(\text{execution}(m.*(..))) \]

**Actions:**
- chooseAIS: int $\rightarrow$ ais
- chooseSaleOrder: int $\rightarrow$ saleOrder
- assemble: int int
- disassemble: int int
- postCheck: $\rightarrow$ bool
- record: ais string int int $\rightarrow$ ais
- getUser: string string
- verifyUser: $\rightarrow$ bool
- getAccess:
- denyAccess:
- controlAccess:

**END** $EXP_{u2}$

Figure 8.59: Module specification of $MOD_{u2}$, Part2
**SPEC_ASPECT** $BOD_{u2}$

**Sorts:**
- string, ais, bool, int, saleOrder,
- mechanic, aisdatabase, authenticationAspect, doaAspect,
- rbacAspect, log Aspect, versioningAspect, coordinationAspect

**States:**
- employedId: string $\rightarrow$ string
- name: string $\rightarrow$ string
- phone: string $\rightarrow$ string
- email: string $\rightarrow$ string

**Joinpoints:** $m$: mechanic
- $\forall m$, execution($m.*(..)$)

**Pointcuts:** $m$: mechanic
- $\forall m$, MROaccessP: execution($m.*(..)$)

**Actions:**
- chooseAIS: int $\rightarrow$ ais
- chooseSaleOrder: int $\rightarrow$ saleOrder
- assemble: int int
- disassemble: int int
- postCheck: $\rightarrow$ bool
- record: ais string int int $\rightarrow$ ais
- getUser: string string
- verifyUser: $\rightarrow$ bool
- getAccess:
- denyAccess:
- controlAccess:

Figure 8.60: Module specification of $MOD_{u2}$, Part3
SPEC_ASPECT $BOD_{u2}$

Advice Axioms: $m$: mechanic, $au$: authenticationAspect

$$\forall m,(B_{pr}(MROAccessP))au.controlAccess()$$

Prescription Axioms:


$$G(q_\theta \rightarrow [assemble()]O(q_\psi))$$
$$G(q_\phi \rightarrow [disassemble()]O(q_\psi))$$

$$\forall au, do, lo, rb,$$

$$(B_{pr}(ReadP())))(I(lo.decryptData; au.controlAccess))$$

$$(B_{pr}(AssDisassembleP())))(I(lo.userActionLogging; au.controlAccess \lor$$

$$(A_{pr}(AssDisassembleP())))(I(lo.userActionLogging; au.controlAccess))$$

$$(A_{pr}(InitializeP())))(I(do.a; rb.b)$$

$$\forall a \in \{\text{generateKey, distributeKey, encryptData}\} \text{ and}$$

$$\forall b \in \{\text{checkRole, distributeRole, checkRight}\}$$

$$(A_{pr}(SearchP())))(I(lo.timeProfiling; do.retriveSIG))$$

$$\forall m, (B_{pr}(MROAccessP))au.controlAccess()$$

$$\forall ad, m.s, do,$$

$$(A_{pr}(InitializeP))do.generateKey(...)$$

$$; do.distributeKey(...)do.encryptData(...)$$

$$(A_{pr}(SearchP))do.retriveSIG(...)$$

$$(B_{pr}(ReadP))do.decryptData(...)$$

$$\forall ad, rb,(A_{pr}(InitializeP))rb.checkRole(...)$$

$$;(q_\theta \rightarrow rb.distributeRole(...)\lor\text{checkRight})$$

$$\forall ad, lo, m,(B_{pr}(SearchP))lo.timeProfiling(...)$$

$$(A_{pr}(SearchP))lo.timeProfiling(...)$$

$$(A_{pr}(AssDisassembleP))lo.userActionLogging(...)$$

$$\forall ad, ve,(A_{pr}(SearchP))ve.checkout(...)$$

$$(B_{pr}(InsertP))ve.commit(...)$$

END $BOD_{u2}$

Figure 8.61: Module specification of $MOD_{u2}$, Part 4
\[
e_{u2} : PAR_{u2} \rightarrow EXP_{u2}
\]
\[
i_{u2} : PAR_{u2} \rightarrow IMP_{u2}
\]
\[
v_{u2} : EXP_{u2} \rightarrow BOD_{u2}
\]
\[
s_{u2} : IMP_{u2} \rightarrow BOD_{u2}
\]
\[
\text{where } v_{u2} \circ e_{u2} = s_{u2} \circ i_{u2} \text{ is satisfied.}
\]

Figure 8.62: Module specification of the \(MOD_{u2}\), Part 5

The aspect "AuthenticationAspect" contains only one advice and its modifier is equal to \textit{before}. Thus, the lines of the weaving algorithm 1 (chapter 6) which are concerned are line 7 to 12. Then, we apply lines 29 and 31 to get the augmented module \(MOD_{w2} = \langle PAR_{w2}, EXP_{w2}, IMP_{w2}, BOD_{w2} \rangle\). \(MOD_{w2}\) (class module specification) is defined by the figures 8.63, 8.64, 8.65, 8.66, and 8.67.

```
SPEC_CLASS PAR_{w2}
  Sorts:
  string, ais, bool, int, mechanic,
  aisdatabase, authenticationAspect, doaAspect,
  rbacAspect, logAspect, versionningAspect
END PAR_{w2}

SPEC_CLASS IMP_{w2}
  Sorts: string, ais, bool, int, saleOrder,
  mechanic, aisdatabase, authenticationAspect,
  doaAspect, rbacAspect, logAspect, versionningAspect
  Actions:
  (one can put here actions of theses sorts)
END IMP_{w2}
```

Figure 8.63: Module specification of the \(MOD_{w2}\), Part 1
SPEC_CLASS $EXP_w^2$

Sorts:
- string, ais, bool, int, saleOrder,
- mechanic, aisdatabase, authenticationAspect, doaAspect,
- rbacAspect, logAspect, versioningAspect, coordinationAspect

States:
- employedId: string $\rightarrow$ string
- name: string $\rightarrow$ string
- phone: string $\rightarrow$ string
- email: string $\rightarrow$ string

Actions:
- controlAccess:
- chooseAIS: int $\rightarrow$ ais
- chooseSaleOrder: int $\rightarrow$ saleOrder
- assemble: int int
- disassemble: int int
- postCheck: $\rightarrow$ bool
- record: ais string int int $\rightarrow$ ais
- getUser: string string
- verifyUser: $\rightarrow$ bool
- getAccess:
- denyAccess:

END $EXP_w^2$

Figure 8.64: Module specification of $MOD_w^2$, Part2
**SPEC_CLASS** $BOD_{w2}$

**Sorts:**

string, ais, bool, int, saleOrder,
mechanic, aisdatabase, authenticationAspect, doaAspect,
rbacAspect, logAspect, versionningAspect, coordinationAspect

**States:**

employedId: string → string
name: string → string
phone: string → string
email: string → string

**Actions:**

controlAccess:
chooseAIS: int → ais
chooseSaleOrder: int → saleOrder
assemble: int int
disassemble: int int
postCheck: → bool
record: ais string int int → ais
getUser: string string
verifyUser: → bool
getAccess:
denyAccess:

Figure 8.65: Module specification of $MOD_{w2}$, Part3
SPEC_CLASS $BOD_{w_2}$

Prescription Axioms:

\[ G(q_a \rightarrow \text{assemble}()O(\psi)) \]
\[ G(q_d \rightarrow \text{disassemble}()O(\psi)) \]
\[ \forall \text{au, do, lo, rb,} \]
\[ (B_{spr}(\text{ReadP}()))(I(\text{do.decryptData};\text{au.controlAccess})) \]
\[ (B_{spr}(\text{AssDisassembleP}()))(I(\text{lo.userActionLogging};\text{au.controlAccess} \lor \]
\[ (A_{spr}(\text{AssDisassembleP}()))(I(\text{lo.userActionLogging};\text{au.controlAccess})) \]
\[ (A_{spr}(\text{InitializeP}()))(I(\text{do.a};\text{rb.\beta}) \]
\[ \forall \alpha \in \{\text{generateKey, distributeKey, encryptData}\} \text{ and} \]
\[ \forall \beta \in \{\text{checkRole, distributeRole, checkRight}\} \]
\[ (A_{spr}(\text{SearchP}()))(I(\text{lo.timeProfiling};\text{do.retrieveSIG})) \]
\[ \forall m, (B_{spr}(\text{MROAccessP})){\text{au.controlAccess}}(\text{ad, m.s, do,} \]
\[ (A_{spr}(\text{InitializeP})){\text{do.generateKey}(...)} \]
\[ ;d.\text{distributeKey}(...);\text{do.encryptData}(...) \]
\[ (A_{spr}(\text{SearchP})){\text{do.retrieveSIG}(...)} \]
\[ (B_{spr}(\text{ReadP})){\text{do.decryptData}(...)} \]
\[ \forall \text{ad, rb,}(A_{spr}(\text{InitializeP})){\text{rb.checkRole}(...)} \]
\[ ;(q_r \rightarrow \text{rb.distributeRole}(...)/\text{checkRight}) \]
\[ \forall \text{ad, lo, m,}(B_{spr}(\text{SearchP})){\text{lo.timeProfiling}(...)} \]
\[ (A_{spr}(\text{SearchP})){\text{lo.timeProfiling}(...)} \]
\[ (A_{spr}(\text{AssDisassembleP})){\text{lo.userActionLogging}(...)} \]
\[ \forall \text{ad, ve,}(A_{spr}(\text{SearchP})){\text{ve.checkout}(...)} \]
\[ (B_{spr}(\text{InsertP})){\text{ve.commit}(...)} \]

END $BOD_{w_2}$

Figure 8.66: Module specification of $MOD_{w_2}$, Part 4

$e_{w_2}: PAR_{w_2} \rightarrow EXP_{w_2}$

$i_{w_2}: PAR_{w_2} \rightarrow IMP_{w_2}$

$v_{w_2}: EXP_{w_2} \rightarrow BOD_{w_2}$

$s_{w_2}: IMP_{w_2} \rightarrow BOD_{w_2}$

where $v_{w_2} \circ e_{w_2} = s_{w_2} \circ i_{w_2}$ is satisfied.

Figure 8.67: Module specification of the $MOD_{w_2}$, Part 5
8.4 Conclusion

In this chapter, we illustrated our approach by means of an industrial case study. This case study entitled "NextGen MRO Instructions Application" has been proposed by Pratt & Whitney Canada [2]. It is a project on which we worked during a training program that we did in this aerospace company in summer 2011. This case study shows that our approach can be applied to a real application. The objective of this chapter is not to present here all the work we did for Pratt & Whitney Canada, but to illustrate our formal framework by using this Pratt’s project. We showed how to:

- identify aspect in an application;
- build algebraic module specification of the components of this application;
- model prevention policy in this application through a coordination aspect;
- weave aspects into the base components by using our weaving algorithm. Recall that our weaving algorithm works since the union operator works.

What we did in this chapter is a partial modular specification because it is a large project which requires an important amount of time and space. We think that this partial specification is sufficient to illustrate our approach. In the future work, this aspect weaving will be done automatically, after we extend the tool MOKA [110]. This extension will take a lot of time, because, we need to rebuild this tool MOKA, by using OCAMLR and introducing aspect concepts.
CHAPTER 9

CONCLUSION

In this thesis, we have presented a formal framework which proposed a solution to
the lack of modular formal reasoning, specification, and verification of aspect-oriented
systems. This framework is based on category theory and algebraic specification due to
their formality, their modularity benefits, and their high level of abstraction. A modu-
lar formal reasoning would significantly help in alleviating the complexity of software
models and application code, as well as reducing development costs and maintenance
time. In this chapter, we summarize the results of the dissertation and also discuss op-
portunities for the perspectives of our work.

9.1 Contributions

During the course of this dissertation, we made the following contributions:

9.1.1 Extension of Algebraic Specification Technique to Aspect Orientation

Algebraic specification is a formal specification approach that emerged in the mid-
70s as a technique to deal with data structures in an implementation-independent manner
[24]. The approach was based on specifying data types in a similar way to that used for
the study of different mathematical structures (e.g. groups, rings, fields, etc.) in modern
algebra. Originally, algebraic specification was intended as a technique for the descrip-
tion of abstract data types. This technique soon grew into a formal specification tech-
nique aiming to cover the whole specification phase within the software development
process. Since then, the research efforts have led to extension of this specification tech-
nique to object-oriented languages. The first contribution of our thesis is the extension
of the algebraic specification technique to the notion of aspect [77, 80, 83, 84, 88]. We
define concrete and abstract syntaxes, and a precise semantics of our aspect language.
We choose to use a loose semantics because it is natural and simple. This semantics
uses algebras and the logic $\mathcal{L}_A$. A semantics of an algebraic specification or a module
specification is defined as a set of deontic models that validate the axioms of this alge-
braic specification or this module specification. To help reasoning about aspect-oriented
systems, the use of formal methods becomes desirable. Formal methods are essential to
support quality, modifiability and reusability by formal concepts for data abstraction and modularity [26].

9.1.2 Logic $L_A$

In Aspect-Oriented Technology, system components can be aspect or class components, which use data structures. Aspect components, class components, and abstract data types are described by aspect, class, and abstract data type module specifications, respectively. As in Wiels’ approach, there are three levels of a system description in our approach:

1. a system is described by modules that are interconnected by morphisms and on which composition operations can be performed;
2. a module is composed of four specifications linked by specification morphisms;
3. each specification consists of a vocabulary part and a set of formulae describing the behavior and the constraints of this specification.

To specify the behavior of software components, we need a logic. This logic has to take into account the societal life of these components which is an important dimension in software system. This societal life has goal to regulate the different interactions between components of a software system. The second contribution is the definition of a logic $L_A$ [78, 82, 86], expressive enough to specify the description and prescription (societal life) behaviors of system components.

The logic $L_A$ includes modalities of three other logics: linear temporal logic (LTL) [55], (first-order) dynamic logic (FDL) [43], and deontic logic (DL) [112]. $L_A$ contains other modalities corresponding to the aspect modifiers Before, After, Around, and InsteadOf. LTL is a simple logic that has been used in many specification works of concurrent and reactive systems [55]. LTL allows us to reason on the time starting at 0 and about the future. It models time as a sequence of states, extending infinitely into the future. FDL is an extension of modal logic, which is intended to reason about computer programs and propositions by integrating the notion of actions (which are computer programs or programs for short). It is worth noting that LTL cannot reason explicitly about actions contrary to FDL. DL is a sub-domain of the modal logic, whose goal is to describe the normative systems. Normative systems model a set of interacting objects or
agents whose behavior is governed by societal norms. Deontic logic can be used in situations where the established rules may be violated. We showed that $L_A$ is sound and complete.

9.1.3 Weaving Algorithm

Weaving is the predominant notion of model combination in Aspect Oriented Software Development [34]. Module interconnections form the architectural structure of a modular system. Four types of interconnection have been defined in [25] and are used in [110]. For each of the four relationships, construction operators have been defined by Ehrlig and Mahr [25] to realize the interconnection associated. We extend the four types of interconnection by adding a fifth kind, the weaving relation. The process of weaving an aspect to a set of base objects (classes) consists in assembling these entities together to produce the final application extended with the behaviors defined in the aspects. The third contribution concerns the definition of the weaving operator corresponding to the weaving interconnection relationship between aspect modules and class modules [79, 84, 88]. This weaving operator uses the notion of colimit, the union operation. The use of colimit concepts is an advantage of our weaving algorithm, because the properties of colimit are what are required by a weaving process in AOSD.

9.1.4 Prevention Policy

Most aspect-oriented verification approaches are based on a detection and correction strategy. Although these detection approaches are relevant for aspect-oriented software reliability, we believe that they are time consuming and costly. It is a good thing to detect and correct system failures, but it is better to first prevent them. The fourth contribution consists of the design of the prevention policy that is used to prevent or avoid undesirable aspect interaction in aspect-oriented systems [85, 87]. This prevention mechanism will prevent most of the undesirable aspect interactions characterized by the fault classes developed in [5]. Our approach is modular, thus making the verification phase modular. This will make the verification phase greatly faster and cheaper. We present a prevention mechanism for four aspect fault types that can cause undesirable behaviors in an aspect-oriented system. These fault types are: Incorrect aspect precedence, Incorrect strength in pointcut patterns, Failure to establish expected post-conditions and Failure to preserve state invariants.

The best strategy of handling conflicts is to prevent conflicts from happening. We can
convince ourselves by making an analogy to medicine, where experts and governments prefer to place more emphasis on measures to prevent disease. The same analogy could be observed in avionics where MRO (maintenance, repair, overhaul) activities put more emphasis on preventive actions than correctives actions for flight quality and reliable aircraft. The spirit of our work is similar to one of Intrusion Prevention Systems.

9.1.5 Application to MRO Applications

Finally, we illustrate our approach by means of an industrial case study. This case study entitled "NextGen MRO Instructions Application" has been proposed by Pratt & Whitney Canada [2]. It is a project on which we worked during a training program that we did in this aerospace company in summer 2011. This case study shows that our approach can be applied to a real application. We show how to:

- identify aspects in an application;
- build algebraic module specifications of the components of this application;
- model prevention policy in this application through a coordination aspect;
- weave aspects into the base components using our weaving algorithm.

9.2 Future Research Directions

For future work, we plan to extend to aspect concepts the tool MOKA developed by the team of Wielu [61, 110] to do some verification tasks. We will name this extension AMOKA. AMOKA will automatically verify the consistency of the categoric modules and some properties on the global system. For those properties which cannot be verified by AMOKA, we will interface AMOKA with a proof tool which can discharge the proofs of these properties. Currently, this tool is in standard ML. Since our framework is aspect-oriented, we need to rebuild it in an object oriented language such as OCAML, by integrating aspect concepts or we can also use an aspect oriented language such as Aspectual CAML [56]. We think that the development of this tool will take a lot of time. After this tool is built, module specifications can be built automatically and the weaving process can automatized.

In this dissertation, we present a prevention mechanism for four aspect fault types (specified in [5, 7, 27]) that can cause undesirable behaviors in an aspect-oriented system. These fault types are: Incorrect aspect precedence, Incorrect strength in pointcut
patterns, Failure to establish expected post-conditions and Failure to preserve state invariants. We are currently working on the specification in the logic $L_A$ of all other aspect fault types presented in chapter 7.

We used union operator to define our weaving operator. Besides, we identify two other interesting ideas which can be used to define the weaving operation. The first one is inspired from the paper [72]. A class module specification can be seen as a client and an aspect module as a server which provides services to this client. Therefore, we can assimilate an aspect-class relationship to a client/server (use) relationship defined in the section 6.3 and then compute the augmented module specification by using the composition operation. The second idea comes from the paper [54]. We can view aspect as a model (or program) transformation. An aspect is a declaration of changes that are to be made to a model or program. Weaving is the process that makes these changes. A program transformation is a function that maps programs to programs (in [54]). We can thus use actualization operation to formalize the weaving process. But, we have to redefine this actualization operation in such way we can actualize a generic module by a module, instead of an actualization of a module by an algebraic specification defined in chapter 6.3. We defer the exploration of these two ideas to future work.
BIBLIOGRAPHY


