AMPLE
Aspect-Oriented, Model-Driven, Product Line Engineering
Specific Target Research Project: IST-33710

Overview of Extensions/Improvements to Existing Implementation Technologies

ABSTRACT
The deliverable describes potential extensions and combinations of the technologies of the partners that could be considered as improvements with respect to identified requirements for variability management. The deliverable analyses benefits and feasibility of these extensions. Finally, a preliminary concept of the implementation platform is described, explaining how the separate implementation technologies could be used together for SPL development.
## History of Changes

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2007-11-13</td>
<td>Initial Version</td>
</tr>
<tr>
<td>0.2</td>
<td>2007-12-03</td>
<td>Results of T3.3 and T3.4 described</td>
</tr>
<tr>
<td>0.3</td>
<td>2007-12-05</td>
<td>Introduction and conclusion for the second chapter</td>
</tr>
<tr>
<td>0.4</td>
<td>2007-12-07</td>
<td>New text on distribution and DSLs</td>
</tr>
<tr>
<td>1.0</td>
<td>2007-12-07</td>
<td>Text about openArchitectureWare integrated</td>
</tr>
</tbody>
</table>
Contents

1 Evaluation of existing AOP and MDD technologies
   1.1 Introduction ................................................. 9
   1.2 Tool Selection ............................................ 11
       1.2.1 Aspect-Oriented Programming ...................... 11
       1.2.2 Model-Driven Techniques ......................... 11
   1.3 A Smart Home Software Product Line .................... 13
       1.3.1 Building Blocks of a Home Automation System ... 13
       1.3.2 A Smart Home Product Line ....................... 14
       1.3.3 A Specific Smart Home ............................ 15
   1.4 Aspect-oriented joinpoint interception ................ 15
       1.4.1 Description ......................................... 16
       1.4.2 Kinds of variability ............................... 16
       1.4.3 Evaluation ......................................... 32
   1.5 Aspect-oriented intertype declarations ................ 34
       1.5.1 Description ......................................... 34
       1.5.2 Kinds of variability ............................... 34
       1.5.3 Evaluation ......................................... 38
   1.6 CaesarJ Feature-Oriented design ....................... 38
       1.6.1 Description ......................................... 38
       1.6.2 Kinds of variability ............................... 41
       1.6.3 Evaluation ......................................... 50
   1.7 HyperJ: multidimensional separation of concerns .... 54
       1.7.1 Description ......................................... 54
   1.8 Dynamic weaving: the DAOP platform .................... 56
       1.8.1 Description ......................................... 56
       1.8.2 Variability dependencies management ............ 57
       1.8.3 Non-invasive dynamic weaving .................... 62
   1.9 JET code generation ..................................... 65
       1.9.1 Kinds of variability ............................... 66
       1.9.2 Evaluation ......................................... 73
   1.10 Conclusions ............................................. 75

2 Extensions and Improvements to Implementation Technologies 81
   2.1 Introduction ............................................. 81
   2.2 Context .................................................. 83
       2.2.1 CaesarJ .............................................. 83
       2.2.2 Steamloom ........................................... 89
       2.2.3 Event-based AOP .................................... 89
2.2.4 openArchitectureWare .................................. 103

2.3 Extensions and Improvements ................................ 105
  2.3.1 Contracts between Features .............................. 105
  2.3.2 Private Inheritance .................................... 107
  2.3.3 Extensible Pointcut Abstractions ....................... 107
  2.3.4 Extension Interface .................................... 111
  2.3.5 Stateful Aspects ...................................... 116
  2.3.6 Dynamic Composition of Classes ....................... 117
  2.3.7 Dependent Classes .................................... 121
  2.3.8 Domain Specific Languages ............................. 123
  2.3.9 IDE Support ........................................... 125
  2.3.10 Expressing Variability in Structural Models ........... 127
  2.3.11 Expressing Variability in Code Generation Templates . 130
  2.3.12 Expressing Variability in Model Transformations ....... 130

2.4 Integration of the Implementation Platform into the AMPLE Process 132
  2.4.1 Integration with the Traceability Framework .......... 132
  2.4.2 Integration with the Architectural Design ............ 133

2.5 Conclusions and Outlook .................................. 134
List of Figures

1.1 Software Product Line Engineering activities (taken from [PBvdL05]) 11
1.2 AspectJ solution for dealing with variation in structure (I) 18
1.3 AspectJ solution for dealing with variation in structure (II) 18
1.4 AspectJ solution for dealing with variation in structure (III) 20
1.5 AspectJ solution for dealing with variation in structure (IV) 20
1.6 AspectJ solution for dealing with variation in data 23
1.7 AspectJ solution for dealing with variation in behaviour 25
1.8 AspectJ solution for dealing with variation in environment 29
1.9 Intertype declaration solution for dealing with variation in behaviour 36
1.10 Smart Home dependency diagram 40
1.11 CaesarJ solution for dealing with structural variability (I) 41
1.12 CaesarJ solution for dealing with structural variability (II) 42
1.13 CaesarJ solution for dealing with variation in data (I) 44
1.14 CaesarJ solution for dealing with variation in data (II) 44
1.15 CaesarJ solution for dealing with variation in behaviour (I) 45
1.16 CaesarJ solution for dealing with variation in behaviour (II) 46
1.17 CaesarJ solution for dealing with variation in environment (I) 48
1.18 CaesarJ solution for dealing with variation in environment (II) 48
1.19 An excerpt of the feature model of the Smart Home case study 60
1.20 Non aspect-oriented component architecture of the Smart Home 61
1.21 Aspect-oriented component architecture of the Smart Home 62
1.22 CAM/DAOP IDE - Architecture Repository 64
1.23 CAM/DAOP IDE - Component&Aspect Repository 65
1.24 JET engine working diagram 65
1.25 Jet solution for variation in structure (I) 66
1.26 Jet solution for variation in structure (II) 67
1.27 Jet solution for variation in data (I) 68
1.28 Jet solution for variation in data (II) 69

2.1 Implementation of a tree structure 85
2.2 Implementation of a colored tree 85
2.3 An example of family polymorphism 85
2.4 Abstract tree 86
2.5 An example of EAOP aspect 90
2.6 An FSP model of a simple e-commerce server 91
2.7 Transformed base FSP 92
2.8 Transformed aspect FSP 92
2.9 Model of the woven system 93
List of Tables

1.1 Mapping between FOA dependencies and DAOP-ADL . . . . . . . 59
1.2 Comparison regarding variation in structure . . . . . . . . . . . . . 76
1.3 Comparison regarding variation in data . . . . . . . . . . . . . . 77
1.4 Comparison regarding variation in behaviour . . . . . . . . . . . . . 77
1.5 Comparison regarding variation in quality . . . . . . . . . . . . . . 78
1.6 Comparison regarding variation in environment . . . . . . . . . . . . . 78
1.7 Comparison regarding variation in environment . . . . . . . . . . . . . 78
Overview

The first part of the document evaluates currently existing Aspect-Oriented Programming (AOP) and Model Driven Development (MDD) tools and technologies regarding variability management in Software Product Lines (SPL), compared to traditional tools and techniques in this context. The goal of this document is to identify what are the novel and positive contributions of AOP and MDD related to variability management in SPL as compared to traditional techniques. Then, strengths and weakness of AOP and MDD will be analyzed and how each one can complement each other will be finally discussed.

In the second part of the document we describe potential extensions and combinations of the technologies of the partners that could be considered as improvements with respect to requirements for variability management that were identified in the deliverable D3.1 [PRG+07]. We explain the benefits and semantics of these extensions as well as how they can be integrated into a coherent implementation technology that better supports SPL development.
1 Evaluation of existing AOP and MDD technologies

This first part of the document evaluates currently existing Aspect-Oriented Programming (AOP) and Model Driven Development (MDD) tools and technologies regarding variability management in Software Product Lines (SPL), compared to traditional tools and techniques in this context. The goal of this document is to identify what are the novel and positive contributions of AOP and MDD related to variability management in SPL as compared to traditional techniques. Then, strengths and weakness of AOP and MDD will be analyzed and how each one can complement each other will be finally outlined.

1.1 Introduction

A Software Product Line (SPL) aims to create the infrastructure for the rapid production of software systems for a specific market segment, where these software systems are similar, and therefore they share a subset of common features, but they also present some variations between them. A main goal in Software Product Line is to, as automatically as possible, to construct specific products where a set of choices and decisions has been adopted on a common model, known as family model, which represents the whole family of products that the Software Product Line covers.

Therefore, for the real success of a Software Product Line, an adequate management of the variations between different products of a same family is critical. Management of variations is mainly influenced by three factors:

1. The placement of variation mechanisms (e.g. compiler switches or design patterns) in the family model.

2. The specification and documentation of the variability of a family of products (using a feature model, for instance).

3. The configuration process, i.e. the construction of a specific product after selecting what specific variants are required. In the ideal case, this configuration process should be completely automatic.

Several tools, e.g. Gears\(^1\) or pure:variants\(^2\), currently support: (1) the definition of family models that contain different variation mechanisms; (2) the definition of models for specifying the variability of that family of products; and (3)

\(^1\) http://www.biglever.com/solution/product.html
\(^2\) http://www.pure-systems.com/Variant_Management.49.0.html
facilities for configuring a specific product as automatically as possible, using the family model as source.

Aspect-Oriented Programming (AOP) [KHH+01] and Model-Driven Software Development [SV06, BBG05] have appeared in the recent few years as new technologies that improves the development of software systems. Both technologies have revealed initially to have important benefits regarding placement and configuration of variations in the context of a Software Product Line.

This document evaluates currently existing Aspect-Oriented Programming (AOP) and Model Driven Development (MDD) tools and technologies regarding variability management, at the implementation level, in Software Product Lines (SPL). They are compared to traditional tools and techniques in this context. The goal of this document is to identify what are the novel and positive contributions of AOP and MDD related to variability management, at the implementation level, in SPL. Then, strengths and weakness of AOP and MDD will be analyzed and how each one can complement each other will be finally outlined.

The process to perform this evaluation is as follows:

1. Representative candidates of AOP and MDD are selected as tools and technologies to be evaluated. Currently, there is a wide range of AOP and MDD tools that can grouped by similarity between them. For instance, AspectJ [KHH+01] and AspectC++ [SGSP02], from the point of view of management of variability, offer the same possibilities, so analyzing both them would result in redundant work. Section 1.2 contains a list of the selected tools and the rationale for that selection.

2. As we need to identify what is required to be evaluated, requirements for variability management are elicited and criteria for analyzing these requirements are also elaborated. Regarding this point, the requirements and the criteria contained in D3.1 [PRG+07] (Section 2.1 and Section 2.2 of D3.1) of AMPLE project will be used. Using the criteria proposed in D3.1 Section 2.1, the new variation mechanisms provided by AOP and MDD will be evaluated.

3. The different AOP and MDD tools will be evaluated using a Smart Home Software Product Line as case study. This case study has been designed to cover all kinds of variability, according to the taxonomy presented in AMPLE D2.2 [SGF+07]. Therefore, it is expected that it covers almost all situations that can appear in a real SPL scenario.

In the following, this part is structured as follows: Section 1.2 contains the tools selected for evaluation and the justification behind each particular selection. Section 1.3 describes the Smart Home Software Product Line use as case study throughout this document. Sections 1.4 to 1.9 analyse and evaluate each one of the identified variation mechanism against each kind of variability identified in the AMPLE taxonomy [SGF+07]. Section 1.10 presents a global comparison of all analysed variation mechanisms and it outlines some conclusions.
1.2 Tool Selection

1.2.1 Aspect-Oriented Programming

We have opted for evaluating AspectJ [KHH01], CaesarJ [AGMO06] and Hyper/J [TOSJH05]. Potential contributions of languages with dynamic weaving, where CAM/DAOP [PFT05] is used as the representative of this category, are also analyzed. AspectJ has been selected because it is probably the most mature and well-known aspect-oriented language. CaesarJ contains advance features for Software Product Lines that are not included in AspectJ. Hyper/J is the most representative language of the multidimensional separation of concerns [TOSJH05] approach. To this respect, we will focus on this document exclusively in the potential contributions of hyperslices (the main difference between Hyper/J and the other aspect-oriented languages) regarding management of variability. Finally, potential contributions of dynamic weaving will be investigated. CAM/DAOP will be used as authors are familiar with this platform.

1.2.2 Model-Driven Techniques

The general scenario of a software product line is divided into two main phases: Domain Engineering and Application Engineering. Each phase is divided into several sub-phases: requirements engineering, design, realisation and testing. At each specific phase one or more models are produced.
According to the schema, we distinguish four possible applications of model transformations:

**Model transformation at the domain engineering level** In this case, the MDD paradigm is applied to the domain engineering process. For instance, domain realisation models are derived from domain design models. This should include the transformation of the variability model when orthogonal variability models are used.

**Model transformation at the application engineering level** The MDD paradigm is applied to the application engineering phase. For example, configuration files for component deployment can be derived from low-level design models. In this case, variants already bound to a variation point in the models defined at upper abstraction layer should be bound in the same way in the derived models. Variant bindings must be resolved in the target model by the model transformation using the information specified in the source models.

**Product derivation** When a specific product configuration is selected, the corresponding application model can be derived by means of a model transformation from the corresponding domain model. For instance, the architecture of a concrete product can be (automatically) derived by means of a model transformation from the architecture of the family of products, after selecting a specific set of variants that must be bound to the different variation points (e.g. see [ZJ06]). This kind of model transformation is specific to software product lines.

**Orthogonal variability to non-orthogonal variability model** When working with orthogonal variability models, variability is not directly reflected in the models (e.g. a component model) that describe the family of applications. Sometimes, software engineers could require visualising variability directly on models. In this case a model transformation can be used to generate a model (e.g. component model) where the different artifacts are marked to indicate variation points, variants or optional parts. This is similar to composing (or weaving) views.

From the four roles identified, we will focus on product derivation. Specifically, we will focus on the use of the code generator JET\(^3\) for defining family models and configuring them. Other currently available code generators, such as XVCL\(^4\) or XPand\(^5\) are based on the same ideas than JET although each of them offer different improved features. However, their evaluation would be quite similar to JET, so they are simply skipped, in order to avoid making redundant work. The reason behind selecting JET is that authors had previous experience handling it.

\(^3\)http://www.eclipse.org/modeling/m2t/?project=jet
\(^4\)http://xvcl.comp.nus.edu.sg/overview_brochure.php
\(^5\)http://www.eclipse.org/modeling/m2t/?project=xpand#xpand
1.3 A Smart Home Software Product Line

The reason for dealing with home automation systems was to get insight into a domain in which applying product line engineering is almost imperative.

Most everyday-life technical devices are controlled by microprocessors. Home automation integrates such devices into a network. The network allows the coordination of the functions provided by different subsystems in order to fulfil complex tasks without human intervention.

The home automation domain tackles as major goals: comfort, security and low costs. Comfort is increased by automating tedious tasks and by an intuitive and well-designed user interface. Security is addressed by identification mechanisms and surveillance equipment like outdoor cameras. Notification mechanisms additionally contribute to security by allowing for immediate reaction. A similar reasoning holds for life safety. Low costs helps to reduce running costs by smart energy management.

The user must be able to access all devices via a common user interface such as a touch screen. In addition, the residents can use Internet applications and mobile computers to control their home from any place.

1.3.1 Building Blocks of a Home Automation System

Sensors and actuators are mechanical or electronic components that measure and respectively influence physical values of their environment. Smart control devices read data from sensors, process this data, and activate actuators, if necessary. For many control and automation tasks a smart control device can act autonomously.

The home gateway is the central server of a smart home. It offers the processing and data storage capabilities required for complex applications. Users such as residents or technicians can access the services offered by the home gateway via different front-ends that interact with the home gateway and provide the user interface.

User management is a necessary component of the home gateway software. Each individual user has different access rights and different preferences with regard to the system functions. This kind of information is stored in the database of the home gateway and can be accessed by other devices such as electronic door locks.

To avoid additional cabling, power-line communication or wireless communication can be used. A realistic home automation system is inclined to employ a heterogeneous network made up of various network standards and various communication media.

The devices connected to the home network can also differ greatly with respect to their functionality and their software and hardware. As a consequence, the software architecture of a smart home must be able to cope with all kinds of networks and technical devices.
1.3.2 A Smart Home Product Line

In this document, we will focus on the development of a Smart Home Software Product Line, with a variable number of floors and rooms (note: the number of rooms per floor is also variable) that offers the following services, categorized in basic and complex facilities.

1.3.2.1 Basic Facilities

**Light management**  Inhabitants must be able to switch on, switch off and adjust the intensity of the different lights placed in a room. The number of lights per room is variable. The adjustment could be performed specifying an intensity value or selecting a predefined value (e.g. ambient, TV watching or reading).

**Doors management**  Doors will open automatically. In addition, access control to several rooms could be required. For authenticating inhabitants, keypad, eye recognition or fingerprint could be used.

**Window management**  Inhabitants have to be able to manage automatically windows. In addition, if the window would have blinds, these should be rolled up and down automatically.

**Heating Management**  Inhabitants must be able to adjust the heating of the house to their preferred value. Heating can operate as in Celsius as in Fahrenheit.

1.3.2.2 Complex facilities

**Smart Heating Management**  The heating control will adjust itself automatically in order to save energy. For instance, the daily schedule of each inhabitant is stored in the SmartHome gateway. In the periods of the day when the house is empty, the heating is disconnected and reconnected with enough time to reach the desired temperature when the inhabitants come back home. If a window is opened or closed, the heating control will take into account it in order to adjust itself. If the opening of a window would suppose high energy consumption, the inhabitants will be notified through the GUI.

**Smart Light Management**  Lights will be automatically switched on and off depending on several factors. If there is no natural light outside the house, the lights will be automatically switched on. Each time an inhabitant enters
in a room where there is not enough, illumination, lights are automatically switch off, unless the room is in the “sleep” mode. This mode prevents automatically management of lights in a room, for instance, because there is somebody sleeping.

**Presence Simulation** In order to avoid burglars, when inhabitants leave the house for a long time, presence can be simulated. There are two non-exclusive options:

**Light Simulation** Lights are switched on and off in order to simulate presence. They are switched on and off using a semi random schema, i.e. the schema varies every day according to random parameters, but lights are not switched on and off without sense. Lights are switched on more often at evening and night, and only occasionally during the day.

**Blind simulation** Blind are automatically rolled up and down, according to semi random schemas. For instance, the blind of the main dormitory is rolled down at night and rolled up in the morning, whereas in order rooms, some nights they are rolled down and some nights they are not.

Each one of these facilities is optional. Each buyer can select the number of facilities he or she desires, although she or he must place devices and facilities at least in three rooms. In other way, the setting up of the Smart Home will be not cost-effective.

Finally, the different facilities can be also acquired in its fault-tolerance version. In this case, special equipment (redundant sensors) and functionalities (fault-tolerance algorithms) will be deployed in addition in order to support this feature.

### 1.3.3 A Specific Smart Home

In order to provide how the configuration process works, a specific Smart Home will be derived. It will have two floors (ground floor and first floor), two equipped rooms in the ground floor, the kitchen and the lounge, and three dormitories in the first floor. Light Management and Door Management are selected for the ground floor. Window management, including Blind Management, is selected for the first floor. Heating is selected for the whole house. Presence simulation with both light and blind simulation is the unique complex function selected. The users desire to manipulate the different devices through a PDA, using the metric system. However, the heating they have acquired operates in Fahrenheit. Touch screens will be placed in the kitchen and the lounge, but not in the dormitories.

### 1.4 Aspect-oriented *joinpoint interception*

This section explores the opportunities, advantages and limitations of using aspect-oriented *joinpoint interception* (also named *pointcut plus advice*) as a mechanism to provide variability at the implementation level, in the context of
a software product line. How joinpoint interception can deal with the different kinds of variability identified in the AMPLE taxonomy [SGF+07] is analyzed. We will use AspectJ [KHH+01] as target language and the Smart Home case study for illustrating our ideas and conclusions.

1.4.1 Description

Joinpoint interception can be considered the most basic aspect-oriented mechanism, which is present in all of aspect-oriented languages, although under different forms and with some slight differences.

An aspect is a mechanism for improving the encapsulation of crosscutting concerns into special modular units, called aspects. An aspect encapsulates a crosscutting concern and provides its functionality through a set of advices (similar to object methods). These advices are not explicitly invoked by the software modules, instead they are triggered automatically. How and when these advices require being executed is specified by means of special composition rules, called pointcuts, which designate loci (in program code) or instants (in program execution) at which advices must be executed. The set of valid points of a program code which can be designated by a pointcut are called joinpoints. Finally, a kind of compiler/pre-processor is the responsible of composing all these pieces of code together as specified by the pointcuts. This composition, called weaving, can be performed at compile time (static weaving), load-time or even run-time (dynamic weaving).

We have selected AspectJ for illustrating this aspect-oriented mechanism as AspectJ can be considered the most mature and well-known aspect-oriented language. In the following, how joinpoint interception can deal with the different kinds of variability identified in the AMPLE taxonomy [SGF+07]. Advantages and limitations of using joinpoint interception will be identified.

1.4.2 Kinds of variability

1.4.2.1 Variation in Structure

A software product line is said to have variation in structure when is possible to derive two different products with a different structure, although they could offer the same functions. For instance, in the case of the Smart Home, it is possible two derive two specific homes with the same facilities, but a different number of rooms, floors or lights per room. These two products differ in number of feature instances.

Structural variation is mainly concerned with the creation of a different number of instances when the product is started. For instance, in the SmartHome case study, creating a room with two or three lights and automatic light management should not be a problem beyond initializing the HouseGateway, since the functions that use these lights should be designed to work with a variable number of objects. We will call to this problem, the problem of the variable constructor.
Additionally, a second problem could appear when the presence or not of one object motivates a change in the behaviour of other objects. For instance, the presence of a wall-mounted touch screen in a room motivates that each time a device placed in that room changes its state, this graphical user interface needs to be notified about the change in the house. We will name this second problem, the problem of the behaviour structurally dependent'.

**The problem of the variable constructor**

In this case, the problem of structural variability can be reduced to the problem of that an object A has associated a variable number of objects of class B (this number could be even zero). In addition, these objects of class B have to be created somewhere. Then, the variability is localized in the constructor of the objects of the class A, which will set as many references to objects of type B as required.

For instance, in the SmartHome case study, if automatic light management is selected, the HouseGateway component will have to keep references to all the light devices deployed in the House. These corresponding light objects need also to be created. As the creation of the light objects as the setting of the references to these lights are variable pieces of code.

The solution offered by the *joinpoint interception* mechanism would be as follows: the code related to the setting of the optional or variable references to objects of class B is outsourced from the constructor of class A and encapsulated into an advice. This advice is associated with a pointcut that intercepts the creation of objects of class A.

This solution extracts out of the creator of class A, code related to the initialization of optional and variable references to objects of class B. Then, the code related to the management of references to objects of one feature (e.g. Light-Management) is encapsulated into a single module, i.e. an aspect, improving readability and maintainability of class A. The aspect will be compiled with the rest of the application if and only if the corresponding feature (e.g. Light-Management) is selected. As the joinpoints where the aspect need to be applied are known in advance for all the products, pointcuts can be fully specified at domain engineering level. On the other hand, advices need to be (manually) written for each product.

The next piece of code shows an aspect that captures the creation of the HouseGateway and adds a room controller for each room of the house. Line 01 shows how the pointcut `addRooms` captures the join point where the HouseGateway is initialized. Lines 02-04 shows the advice that is executed when the constructor of HouseGateway returns. Two RoomController objects are created by the advice and added to the HouseGateway list of room controllers.

It could be also the case that the newly created objects of type B (e.g. Room-Controller in the previous case) require also be associated with a variable number of objects of type C, so the previous solution has to be applied again, but this time intercepting the creation of the objects B and executing a different ad-
vice each time an object B is created. This advice, as before, is responsible for adding a concrete number of objects of type C.

For instance, the previous example could be extended adding the possibility to have light controllers. The HouseGateway component has to keep references to all the rooms controllers like in the previous example, and each RoomController has to keep references to the light controllers it contains. The AspectRooms aspect, which manages the addition of room controllers when the HouseGateway is created, is the same one we used for the first example. The AspectLights aspect works in a very similar way. The addLights1 pointcut captures the join point where the constructor of first RoomController is called. Line 07 shows how the pointcut addLight1 selects specifically the creation of light controllers for the Room 1 (r.getRoomNumber()==1). This differentiation is necessary because the light controllers to be added depend on the concrete RoomController object. This implies we need to construct a pointcut and an advice for each Room object. Lines 08-10 show the advice that is executed when the constructor of the RoomController number 1 returns. In lines 09-10 the light controllers for that room are created and added to the RoomController.

As the pointcuts as the advice need to be written manually for each application at the application engineering level. It is also necessary to create a pointcut and an advice for each concrete object, i.e. for each room with a different number.

Using the solution of Figure 1.3, the number of pointcuts and advices would grow quickly, proportionally to the number of dependencies between variable associations in the family model. In addition, all these joinpoint interception are required to be written manually during application engineering, which makes
this solution not very attractive for dealing with this kind of variability. Finally, we have considered until now all the variable objects need to be created by the advices, but it can be also the case in which these objects are already created. i.e. an object B is associated with a variable number of objects of type C, and another D object is also associated with a subset of the objects of type C related with B. Therefore, the advice that intercepts the creation of the object D must not create the subset of objects of type C, as they are already created. Instead, such an advice must recover references to these objects of type C and assign them to the object D during its initialization. Again, these advices are required to being written manually during application engineering. They can also use a configuration file that specifies how many object and with which parameters need to be created. This only shifts the problem of writing a large amount of practically identical information from the advice body to a configuration file.

One problem for adopting this solution is that we need to ensure that the advice associated to the creation of the object D (e.g. LightManagement) is executed after the execution of the advice associated to the creation of the object B (e.g. RoomController), otherwise, the required subset of objects of type C (e.g. LightController) would not be created. AspectJ does not provide any mechanism for declaring the ordering of advice execution when the advices are specified in different aspects.

For instance, in the SmartHome case of study an optional light management controller is added to the HouseGateway class of the previous example. The LightManagement object needs references to the LightControllers of the SmartHome. The aspects used to create the room controllers and the light controllers are the same ones used in Figure 1.3 example. The LightManagement creation and the addition of references to the light controllers are done by the AspectLightManagement aspect. When the HouseGateway constructor returns, there are two advices that could be executed; the advice in the line 02 of the Figure 1.3, that creates the room controllers objects and the advice in line 02 of Figure 1.4, that create the LightManagement object. This is the situation where the precedence problem appears and there is no a direct way to ensure which advice will be executed first. The problem of precedence is solved by the pointcut in the line 01 ensuring that the two room controllers have been created (\( g \text{.getRooms()}.size()>1 \)) and therefore the rooms and light controllers have been already created. Line 03 creates the LightManagement object and adds it to the HouseGateway. Lines 04-08 search in all the RoomControllers of the HouseGateway to add the references of the LightControllers into the LightManagement.

The problem of the behaviour structurally dependent

---

6The precede clause does not work if advices are defined in different aspects. Thus, the only thing engineers can do is this case is to cross their fingers to expect the AspectJ weaver behaves as they want.
The problem of behaviour structurally dependent appears when a class A (e.g. LightController) needs to modify its methods depending on the structure of the program. This means that the behaviour changes if there is a specific instance of a specific call (e.g. a WallScreen instance exists in the same room of a LightController).

Using joinpoint interception, the methods that need modifications or additions of the class A if a certain object B exist are selected by a pointcut. The advice associated with such a pointcut will override or modify the methods of class A if and only if the object B exists, which is checked at the beginning of the advice.

For instance, in the Smart Home case of study, the LightController objects needs to notify to the WallScreen object of the same room if the intensity of the light is modified. However this behaviour is only necessary in case a WallScreen exists for the same room where the LightController is placed. An aspect that captures the methods dealing with intensity changes in the LightController class is constructed. In line 03, it is specified that the calls for setting a new intensity value must be captured. Then, an advice checks if there is a WallScreen instance for the room where the LightController object is, and if so it calls the adjustLight method of the WallScreen to notify the change.

Contributions

In traditional Software Product Lines, the problem of the variable constructor is solved using precompiler switches, parameters, templates, design patterns, and in the worst case, manually. When the number of optional associations increases as well as the dependencies between these optional associations,
constructors of certain classes can become unreadable and unmanageable. For instance, as the numbers of devices that can be included in a SmartHome increases, the HouseGateway will have to handle the initialization of lights, light switches, dimmers, heaters, blind rolls, windows, thermostats, fire detectors, smoke detectors, and so on. Additionally, the creation of certain objects will depend on the creation of other objects, for instance, dimmers and light switches may depend on the lights they control. Therefore, the HouseGateway constructor can be a complex piece of code.

However, this complex piece of code can be modularized simply outsourcing its different parts to external functions, which can be then included or excluded from the constructor using just precompiler switches or macros. Therefore, the only advantage that provides the aspect-oriented solution is some degree of obliviousness, as using aspect-orientation, the constructor could be rather unaware of how these optional references are created and initialized.

The manual creation of each advice does not also improve the traditional solutions. It can be state that these advices could be codified in a generic way and use then, for instance, configuration files that specify how many instances and which with data have to be created. However, this is nothing different from the traditional solutions; the only difference would be the configuration file is read from an advice instead of the creator itself, which it is not a noticeable improvement.

Finally, for the “variable creator problem”, the solution provided by joinpoint interception does not also improve management of variability in the context of software product lines either. Features will be present in a specific product or not depending if certain aspects are included or not in the compilation of the product. This means some kind of variable make or ant file need to be created, which implies the use of some kind of parameterization in this file, which will be not different to the use of precompiler switches.

A potential advantage of using the joinpoint interception technique for structural variability is that this solution allows the modification of a constructor without changing its code. However, the use of aspects in this way could be considered a kind of patching, which, according to several opinions, is not the better way for using aspects.

1.4.2.2 Variation in data

Variation in data refers to the possibility of deriving two specific products from the same SPL which operate with different input, output or intermediate data.

We distinguish basically two different kinds of variation in data:

1. The data types are not the same. For instance, a heater works with temperatures expressed as integers, but the thermostat sends temperatures measured as float numbers; or a video surveillance system has a camera that can send colour or gray scale images.

2. The data types are the same, but they are semantically different. For instance, heaters in the SmartHome SPL can accept float numbers as
temperature values, but these float numbers can express temperatures either in Celsius or Fahrenheit.

We comment on solutions for both approaches below.

**Different data types**

The solution will be to write an aspect with an advice that performs the corresponding data conversion. For instance, an aspect advice could convert colour images into gray scale images, or convert float to integers. The main advantage of using this technique is that the communicating entities does not need to be modified, thus, we can call this, *non-invasive adaptation*. Otherwise, this problem is often solved by current compiler technology: integer values can be assigned to float variable, float numbers can be assigned to integer variables if we use castings or we set up a certain compiler option. Most complex data type compatibility, such as between colour and gray scale images, can be achieved using object-oriented inheritance plus polymorphism, as for example, making colour images a subclass of gray scale ones. Finally, data conversion that requires data manipulation, as changing the ordering of an array from ascending to descending order, can be achieved by simply calling to static methods that carry out these conversions. We would like to point out that when a conversion is not feasible using object-orientation (e.g. conversion of black and white images into colour images); the conversion will not also be possible using aspect-orientation. Aspect-orientation can specially help when this conversions has a crosscutting nature.

**Semantically different data types**

In this case, the data types are the same, but they express different values because they represent different elements. For instance, a float could represent a money amount, but this can be expressed in euros, pounds or dollar, a temperature can be represented as an integer, but this integer can represent a value in Celsius or in Fahrenheit. In this case, an aspect would be responsible of executing the corresponding semantic conversion where required. This solution does not differ from the previous one, and it has the same advantages and drawbacks.

For instance, in the SmartHome case of study, the RoomHeatingController communicates with the HeatingTemperatureSensor controller to adjust the room temperature. The chosen TemperatureSensorController has been developed to work in Celsius degrees and the heating control to work in Fahrenheit degrees. The aspect AspectTempSensor is used to adapt the getTemperature method of the HeatingTemperatureSensor class, converting as required the return value.

The pointcut trasformTemp (line 01) captures the joinpoint where the getTemperature method of HeatingTemperatureSensor is called. The advice of line 02-03 overrides the getTemperature method and converts the temperature of
Figure 1.6: AspectJ solution for dealing with variation in data

the HeatingTemperatureSensor object to Fahrenheit degrees before returning the value.

As it has been already commented, aspect-orientation does nothing here that can not be done using object-orientation, which has as one of their major contributions dealing with variability in data by means of inheritance and polymorphism. In addition, other solutions as castings, conversion libraries and generics might be employed.

However, these solutions require the updating and modification of the source code, which can be avoided using pointcuts and advices. Thus, the main motivation for using aspect-orientation is that, we would not need modify the source code of the communicating entities in order to create the proper castings or adopt an object-oriented solution for a certain reason (e.g., these communicating entities are prebuilt or legacy modules that we can not alter).

Another reason for using joinpoint interception is for performing this data conversion is to improve the readability of a piece of code with a high degree of variability in data. This piece of code might be plenty of optional castings, that will be present or not depending of if certain features are selected or not. Aspect-orientation allows the extraction of all the code related to dealing with variability in data outside the aspect and in separate files, improving its understanding. In this case, the selection of certain features would motivate the inclusion or exclusion of certain aspects from the compilation unit, instead of invasively modifying the source code of the application modules. This improving in code understanding may also probably benefit maintainability and scalability, since it provides some kind of positive variability.

Finally, another advantage of aspect-orientation relies on quantification as an instrument for dealing with crosscutting data conversions. A unique pointcut might designate multiple places where a data conversion is required (e.g all the setter/getter that refers to temperature data), avoiding the updating of several methods. Using joinpoint interception, only one pointcut and one advice need to be written. Thus, aspect-orientation brings some advantages when dealing with crosscutting data conversions, which would require to perform the same conversion in multiple program code places.

However, quantification in aspect-oriented programming often relies in the adoption of certain name conventions. If these conventions are not followed, quantification effectiveness might decrease highly.
1.4.2.3 Variation in behaviour

Variation in behaviour refers to the possibility of deriving two specific products from the same SPL which present different behaviour.

We distinguish three different kinds of variation in behaviour (which are not mutually exclusive):

1. Variation that implies the addition of new software modules (e.g. light controllers) which provides new functionalities (e.g. automatic light management).

2. Variation in the implementation of a service (e.g. different LightController classes can implement a same ILightManagement interface).

3. Variation in how the different services are coordinated (e.g. smart heater management controlling the window and heater management).

We comment on this kind of variabilities below.

Addition of new components with new functionalities

The first kind of variation in behaviour requires mainly the addition of new classes (e.g. LightController), which support the new functionalities. This implies that the previously existing (base) classes will require managing references to these elements and should be prepared for that. This problem has been already considered, in the “variation in structure” section (section 1.4.2.1), so we refer the interested reader to it.

Variation in the implementation of a service

In this case, we distinguish another two different kinds of variations in the implementation of a service: (1) variations that affect to only one service (e.g. changes in the adjustValue method of the LightController class); (2) variations that affect to several services in the same interface or even several interfaces (e.g. all value settings of any controller must be persisted after the corresponding setter are executed).

For the first situation, variation in the implementation of a service, the problem can be solved just using object-oriented techniques such as abstract classes, interfaces, inheritance and polymorphism. It would be enough to create an interface, or abstract class, that different concrete classes implement in a different way. Also, a subclass can inherit from a superclass and overrides the corresponding method.

For the second situation, using object-orientation, multiples method should be overridden adding a similar piece of code, i.e. there is a crosscutting piece of code. Therefore, it makes sense to use aspect-orientation for dealing with this kind of variation, as aspect-orientation aims to solve this kind of problem. In this
case, an aspect will intercept the execution of all these services and it will add the corresponding extra-behaviour.

For instance in the Smart Home case of study the LightController class implements the interface ILightController to define three methods which change the light intensity (e.g. increaseIntensity(delta), decreaseIntensity(delta), setIntensity(concreteValue)). If a GUI needs to track these values, each time one of these methods is executed, a notification to the GUI requires being sent. Using just object-orientation, this would require creating a new subclass for Lights and redefining each method that changes the intensity in order to add the code for sending the notifications. Using aspect-orientation, this crosscutting piece of code is encapsulated into an advice that is executed each time a method that modifies the intensity is called. Line 01 describes the pointcut that captures the execution of any method of LightController which name ends with “Intensity” call(void *Intensity(..)). Lines 02-03 implement the advice that will be executed when the method is finished. Line 03 sends a message to the GUI with the new intensity. In the example it will only print it in the screen.

**Variation in how the different services are coordinated**

In this case, the protocol that governs the interaction between different software entities changes, either because we are adding new functionalities on top of these components, or because we are using an alternative coordination protocol.

The coordination protocol that governs or orchestrates the interaction between different software modules crosscuts these modules [FS07]. Each software module that is governed by this protocol has to perform two tasks related to two different issues: (1) computation, i.e., the execution of its core functionality; and (2) coordination, i.e., the management of interactions with other entities in agreement with the coordination protocol. The crosscutting nature of coordination and the benefits derived from separating coordination from computation has been acknowledged by the coordination community, and several exogenous coordination models [Arb04, CPRC03, LLW06, MHSA99] have been proposed in order to overcome these shortcomings.

Aspect-Oriented Programming has demonstrated to be an adequate technique for implementing coordination protocols in a modular fashion, separating coordination from computation. The basic idea is that a coordination aspect interceps the communication between software modules being able of...
filter, redirect, broadcast, queue and so forth, messages, according to a certain coordination protocol. Aspect-Oriented Programming has also demonstrated to be a suitable technique for implementing exogenous coordination models.

The changing of the way in which a set of software modules interact because we are adding new functionalities on top of these modules, implies we need to superimpose a new coordination protocol on top of these modules. The problem can be solved using the technique presented by Fuentes and Sánchez [FS07], which consists in to write an aspect that intercepts, by means of pointcut expressions, those messages that makes the coordination protocol to react, and specify the logic of the coordination protocol in the aspect advices. The second problem, the changing of the coordination protocol, can be solved by simply changing the aspect that encapsulates it.

A specific example of how the auction protocol for an Auction System can be implemented and easily changed from the public English auction to the private one-bid style using aspect-orientation can be found in [FS07].

Contributions

Since the methods used to solve this variation are based on the ones used in section 1.4.2.1 the benefits of them against a standard implementation are very similar depending of the specific problem to be resolved.

For all the cases of variability in behaviour, advantage of using aspect-orientation if that the changes are performed externally to the software modules code, which is specially an advantage is the source code of these modules is not available. This happens when these modules are off-the-shelf components or legacy code that we can not, or we should not, update. Optional components could be added externally using aspect but the classes have to contain optional references to those components or generic type component references.

In the case of redefining existing behaviours, aspect-orientation brings benefits when these changes have a crosscutting nature, i.e. it affects to multiples services of the same or different modules. This is not surprising as aspect-orientation objective is to improve how to deal with this kind of crosscutting issues. The advantages relies on that: (1) using encapsulation this changes are uniformly applied to several software modules at the same time; (2) this piece of crosscutting code is well-encapsulated into a single piece of code, improving the system modularisation and with the already well-known benefits regarding system maintenance and evolution [Par72, Cla02, LB06, GBF+07]; and (3) whenever obliviousness can be achieved, those modules whose behaviour has been varied have not any dependency regarding this behaviours.

Finally, regarding changes in the coordination protocol that governs several software modules, as all the logic of the coordination protocol is encapsulated into an aspect instead of being spread over all the software modules that are coordinated, the impact of changes related to the coordination protocol is reduced. Additionally, as there are not dependencies from the software modules that are coordinated respecting the coordination protocol, this change can be
more easily performed. In the other direction, due to the current syntactic coupling of aspect regarding the modules they crosscut, a change in the signature of the messages that are coordinated will affect to the aspect that encapsulates the coordination protocol.

Selection of the variants is carried out by compiling or not compiling the aspect which implements the variation. The advice bodies of these aspects can be fully specified at domain engineering level, but the pointcuts could be required being specified at the application engineering level, since a variation in behaviour can only be applied to a specific feature instance, for instance to the light “3” of the first floor of the John’s house.

1.4.2.4 **Variation in quality**

Variation in quality refers to the possibility of deriving two specific products from the same family which present different quality attributes (e.g. a different performance or different levels of security).

These variations in quality can be due to different reasons: (1) a variation in the internal implementation of one or more methods of one or more classes (e.g., implementations of a method with different performance); (2) a variation in the quality attributes (e.g., different security schemas) that are applied over a software module.

Both cases can be managed as a kind of variability in behaviour described before. In the first case, object-oriented solutions could be used, in the second case, due to the crosscutting nature of the quality attributes, the aspect-oriented solution is recommended. The only difference with variability in behaviour is that, the details of this kind of variations are usually quite technical and not of interest of end-users.

For instance, in the Smart Home case of study, the SmartHeatingManagement class implements several algorithms to control the house heating system. One of this algorithms could be a save energy mode to control the aperture of the windows and the heating control in order to achieve the desired temperature with the minimum energy waste. This method could be reimplemented using aspects by capturing the method to achieve less energy consumption or use less computational resources increasing the quality of the system.

1.4.2.5 **Variation in environment**

Variation in environment refers to the possibility of deriving two specific products from the same family that are deployed in environments with different characteristics.

We identify three different kinds of problems we need to address:

1. The software modules (e.g. classes or components) that comprise a specific product are distributed into different kind of nodes. For example, a SmartHome GUI component could be deployed in lightweight devices, such as PDA, or a common PC.
2. The software modules (e.g. classes or components) that are part of a specific product could be distributed on nodes differently. For instance a typical web information system comprise of web interfaces, a business layer and a database. This information system can be deployed using a two-tier schema (the business layer and the database layer are deployed in the same server) or a three-tier schema (the business layer and the database layer are deployed in different servers).

3. A specific product deals with different external services (e.g. a SPL could require a payment platform, such as CreditCard service. Thus, depending on the selected service some variations could emerge, such as different method signatures in the interface of the services or different access protocols).

**Variation in the node type**

In this case, the family model is most likely to have different versions of the same component depending on the kind of node where this component it is going to be deployed. For instance, the most used solution will be to have two versions of the SmartHome, one for being deployed in common PCs, and the other one for being deployed in lightweight devices, such as PDAs. Additionally, the normal solution is to have one component for each kind of PDA, as there are often slight differences between them. Therefore, the problem has two different faces: (1) implementing several components that share functionality and are similar with the minimum effort and as good maintenance, evolution and reusability as possible; and (2) managing alternative references to the changing components.

For the first part of the problem, it is a case where aspect-orientation usually provides few benefits, as the required changes are not crosscutting and very well-localized, and simply inheriting and overriding methods or using code-generators templates is enough. In this case, the *joinpoint interception* technique becomes often a simple patching technique.

In the second case, aspect-orientation could be useful to externally manage the alternative references to the changing components. A solution is to instantiate each alternative component (e.g. LightweightGUI or HeavyweightGUI) using an aspect. When the constructor of the class that contains the alternative reference (e.g. LightController) is called, an aspect captures the call and instantiate the component to the desired class type. In order to do that the alternative components have to implement a common interface or inherit from the same abstract class. The alternative reference will be of the common interface type or the common abstract class type.

For instance, in the Smart Home case of study, the HouseGateway has two different kind of graphical user interfaces, one for PDAs and one for common PCs. Each interface corresponds to an option in the feature model. Both components implement the interface IGUI. The HouseGateway class has a reference to an IGUI object, which is initialized by means of aspects. A pointcut (line
01
capture the call to the constructor of HouseGateway. The lines 02-03 implement the advice that is executed when the constructor of the HouseGateway class returns. In line 03, the light GUI component is instantiated to the concrete class type to use in this case.

**Variation in the deployment configuration**

When the way in which software modules are distributed among different nodes changes, additional changes related to the way in which these software modules communicate need also to be performed. For instance, let us suppose two components are deployed in the same node. Then, they can communicate directly using normal method calls. If these software modules, in another product of the SPL, are distributed in different nodes, they need to use some kind of remote procedure call or middleware technology in order to be able to communicate between them. A solution is to use aspects to capture the communication among the components, and in case they are in different nodes redirect the communication as required, using an adequate technology for dealing with remote communications, such as Java RMI or CORBA. Using *joinpoint interception* is possible to add the crosscutting piece of code that redirects these communications.

One benefit of using aspects here is that it is possible to keep the direct communication between components when they are in the same node and move them to other nodes without changing internally the code of the components, although similar benefits can be achieved using middleware platforms such as CORBA (nevertheless, middleware platform often requires some kind of invasive changes in the software modules involve in remote communications).

**Variation in the external services used**

Changing an external service used by a software module can be as easy as changing a reference, in case the interfaces of the alternative services were the same (as in signature as in protocol) and no adaptation were required. Techniques for changing references using pointcuts and advices were already commented in the subsection regarding variation in structure.

In case the interfaces of two alternative services do not were identical, it implies some kind of adaptation is required when switching from one service to the
other one. The mismatching between software modules interfaces is solved using \emph{component adaptation} [YS97]. This technique consists in writing a piece of code, called \emph{adapter}, which serves for matching a pair no-previous matching required/provided interfaces. The use of aspect-orientation brings some benefits to the implementation of adapters. Using aspect-orientation, a pointcut is responsible of picking up those points of the communication between two components where a mismatch appears, and the logic for the adaptation would be encapsulated into a pointcut.

Mismatching between provided and required interfaces can appear due to signature mismatching (i.e. differences in methods names, arguments types and so on) or protocol mismatching (i.e. differences in the ordering in which the messages of the interfaces must be invoked).

For instance in the Smart Home SPL, the SmartHeaterManagement class has a functionality for automatically detecting the gas tank is getting empty and request a new one to the gas provider, through a remote service. Different gas providers use a service interface with a different protocol (ordering of messages). An aspect is then created to capture the calls from the SmartHeaterManagement class to the gas provide service and reorder the messages according to the protocol interface of the specific gas provider being used.

A particular problem in this case is that the source code of the external services is not available, so any aspect-oriented technique that requires source code modification must be used carefully. In the case of AspectJ, we must ensure the weaving process does not require accessing the source code of the external service. This problem can be solved by simple using an aspect-oriented language with a non-invasive weaving mechanism, such as DAOP [PFT05].

More examples and in-depth discussion about this issue can be found in [FS05, ATB96, VW01, RC02, PFA03, VCV+04, DYBJ04, RK04].

\section*{Contributions}

Depending on the kind of variation in technology, \emph{joinpoint interception} can be a suitable technique for dealing with it or not. When several versions of the same component need to be develop, each one for a different type of node (e.g. PC, PDA or mobile phone), aspect-orientation does not offer any special feature beyond a more or less elegant solution for patching the code. In this case, model-driven technologies offer a more elegant and clean solution.

When the variation is related to the deployment configuration of the different SPL components, \emph{joinpoint interception} gives the possibility of moving components to different nodes with a certain degree of transparency, although the advantages are more or less the same than using current middleware technologies, such as CORBA, DCOM or Java RMI.

When the variation is related to the external services a product uses, \emph{joinpoint interception} offers more interesting possibilities, since they are able to modify the interactions with the external service transparently to the product, allowing the creation of external adapters. If the aspect-oriented language would offer
a non-invasive weaving schema, such as DAOP [PFT05] or JAsCo [SVJ03], it would be even possible to construct non-invasive adaptors that would not require the internal modification of the adapted components. Therefore, aspects are able to perform oblivious adaptations of these services, which provide several advantages as compared to traditional techniques, such as the wrapper pattern, as for instance, avoiding the object schizophrenia problem [Tru04].

AOP could seem very similar to container technology, but there are some noticeable differences: (1) services offered by containers are not managed homogeneously as in AOP; (2) available services of containers are not modifiable or extensible, as they are hard-coded in the component platform and therefore set of available services is fixed, being developers not able to add new services.

Variability is acquired by compiling or not the corresponding adaptation aspect, which promotes the encapsulation of variants into separate units.

1.4.2.6 Variation in technology

Variation in technology refers to the possibility of deriving two specific products from the same family of products which are constructed using different software technologies or abstractions. By software technology or abstraction we mean different programming languages, different programming techniques (e.g., recursion vs iteration) and so forth.

We have identified two potential sources for variation in technology at the implementation level: (1) changes the programming language; and (2) changes in the set of abstractions, primitives, techniques or guidelines that are used to construct the software (e.g., synchronous messages or events).

In the first case, there is little that the mechanism joinpoint interception can do, as the only possible solution is often to have the system replicated once by each target language.

In the second case, there is a wide range of different possibilities situations of different nature, so it is practically impossible to give a general solution that satisfy all of them and also to group the different situations in well-defined categories.

For instance, let us suppose there is governmental organization responsible of certifying quality and correctness of software programs according to a well-defined criterion. One of the points of this criterion is that there should not be goto-like sentences in an application code, as it is considered a source of potential errors. The organization runs an automatic code analyzer to ensure this and other requirements. Thus, in case, C is used as programming language, the use of switch statements would be prohibited since these code blocks contain break statements that are considered a kind of goto. In order to obtain this certification, a software company replaces all the switch statements for if code blocks. Nevertheless, for the software company, this change makes the code less efficient and hard to maintain. For this reason, the company decide to have two products, one non-certificated with switch statement and the other one certificated, but with if statements.
Aspect orientation could be useful when changes required for the solution implies the addition or a crosscutting piece of code. For instance, the communication between different software modules can be changed from an endogenous model to a exogenous model using aspect-orientation. Readers interested in this subject can refer to [CSZ04].

1.4.3 Evaluation

This section contains a general evaluation of the *joinpoint interception* mechanism as a mechanism for implementing variations.

Expressive Power

*Joinpoint interception* is mechanism that allows the encapsulation of a piece of code that represents a variation into a well-localized module called *aspect*. This piece of code can be added to multiple places of an application code using *quantification*. This piece of code, i.e. the aspect, can refine or override the application code where it is applied. The granularity of these refinements/overridings relies on the *joinpoint model* of the aspect-oriented language selected. To this respect, AspectJ offers a wide joinpoint model, allowing the interception of a considerable number of points in the execution of a program, richer joinpoint models can be found in the literature, such as in the ALPHA language [OMB05].

In addition, the quantification power of aspect-oriented programming allows the application of these refinements/overridings to a set of software modules at the same time, contributing to the proper encapsulation of crosscutting concerns.

Finally, and not least important, aspect-oriented languages provides some kind of reflective actions (e.g. thisJoinpoint) that helps to parameterize the crosscutting piece of code they encapsulates in function of certain elements of the context they refine override.

The *joinpoint interception* technique can be applied to variations in structure variability, in data, in behavior, in quality and in technology, where the main benefits has been identified in the management of crosscutting variations in data, behavior and quality. For variation in structure, aspects can help to solve the problem of the structurally dependent behavior and in the case of variation in environment, the main advantage of joinpoint interception is that it eases the construction of adapters. Finally, and not least important, *joinpoint interception* helps to separate coordination from computation contributing to the implementation of some kind of variation in behavior.

Binding Model

Depending on the kind of aspect-oriented language, the binding model can be performed at compile time, such as in AspectJ [KHH+01], at load-time, such
as in Aspectwerkz\(^7\), or even at runtime, such as in JAsCo [SVJ03]. In the last case, variants are also available at runtime, allowing the system to be extended with new variants not originally considered at design/compile time.

**Validation**

One of the advantages of aspect-oriented techniques is that variants are statically types and type conformance can be checked.

**Modularity**

Aspect-orientation contributes to modularization as it allows the proper encapsulation of a crosscutting piece of code in a single module, avoiding scattering and tangling and therefore increasing cohesion and reducing coupling and leading to a better maintenance, evolution and reusability [LB06, GBF\(^7\)07].

Secondly, whenever obliviousness can be achieved, core assets are unaware of the variants that are applied over them. The same statement in the opposite direction is not always true due to the current syntactic coupling between aspects and the software modules they crosscut and changes in the core assets could have an undesired ripple effect in the product variants. Therefore, it can be concluded core assets are independent of the variants applied, but variants are in some degree dependent of core assets, which reduces their reusability.

Core assets can declare explicitly their interfaces, but the definition of aspect interfaces is often weak in most of the aspect-oriented languages. In general, aspect-oriented languages lacks of a well-defined and established modular definition of aspects that eases aspect reusability and extensibility, as for instance, by means of inheritance. However, certain languages, such as CaesarJ [AGMO06], contains noticeable improvements regarding this point.

Aspects are also unaware of other aspects applied to other modules which lead to interaction problems if two conflicting aspects are applied on the same joinpoint [DFS02a]. The techniques for solving aspect-interactions issues are not well-defined at all in the aspect-oriented community, being one of the current aspect-oriented challenges. In general, each particular aspect-oriented language defines its own mechanism to cope with aspect-interactions.

Finally, it should be noticed that the proper encapsulation of any crosscutting concern into an aspect could be not always possible, existing some crosscutting concerns, such as complex transactions mechanism, whose proper encapsulation into aspects remain a challenge for the aspect-oriented community [KG06].

**Performance and usability** The efficiency of an aspect-oriented application is mainly influenced by how the weaving process is implemented. In the case of AspectJ the use of aspects should not imply a serious performance overhead, as the weaving performed by the AspectJ compiler is static and invasive,

\(^7\)http://aspectwerkz.codehaus.org/
the woven code should be of a performance similar to the non-aspect-oriented version, as in memory as in time. In case of dynamic weaving, different penalties will be introduced, although languages such as Aspectwerkz or JAsCo do not have a noticeable decrease of performance in spite of performing dynamic weavings (load time weaving in case of Aspectwerkz, and runtime weaving in case of JAsCo).

Aspect-orientation introduces new constructions and abstractions which implies a certain learning process by developers before being used. However, these abstractions, as compared with other similar technologies, such as reflection, could be considered as easier to learn and use [Sul01].

Aspect-oriented techniques are able, using the new constructions provided, to modify the normal control flow of a program, which generates new issues to be taken into account with the application code must be tested, as new failures and exceptional situations could be introduced due to the use of aspects. To this respect, new methodologies for testing aspect-oriented programs are appearing [XX06], although they are still in its early phases.

The tool support for aspect-oriented languages varies widely, since very early prototypes to mature languages. In the case of AspectJ, it can be considered a mature language, well-integrated into the Eclipse platform, and with a set of additional facilities, such as the crosscutting viewer or a debugger.

1.5 Aspect-oriented intertype declarations

This section describes intertype declarations, a construction that can be found in most of the Aspect-Oriented languages, can help to manage variability in the context of a Software Product Line.

1.5.1 Description

An intertype declaration is an AspectJ mechanism that combined with joinpoint interception allows the modification of some parts of the structure of the classes that comprise an application. Using intertypes declarations, it is possible to add new methods to a class, add new attributes or change inheritance relationships. An intertype declaration must be specified inside an aspect.

1.5.2 Kinds of variability

1.5.2.1 Variation in structure

The “problem of the variable constructor” (see section 1.4.2.1) can be reduced to the problem of that an object A has associated a variable number of objects of class B. The variability is localized in the constructor of the class A, which needs to create a variable number of instances of object B.

Using intertype declarations, the solution would be to declare a new constructor for the class A with the code necessary for creating the exact number of
instance of class B as required. However, the signature for this new constructor can not be the same as the other constructor already exiting in class A, since, using intertypes, it is not possible to introduce a new method whose name and signature are the same as a already existing one.

Any special benefit has been found in intertype declarations as compared to joinpoint interceptions for dealing with variability in structure.

In the problem of the “behaviour structurally dependant”, a class A needs to modify his methods depending of the structure of the program (see section 1.4.2.1). No direct solution using just intertypes declarations have been found for this case. It would be possible to combine intertype declarations and joinpoint interception using the mechanism specified in section 1.4.2.1. In this case, an pointcut is executed on the method that need to be augmented with a behaviour structurally dependant, and this behaviour could be placed in a inter-type delcaration, but there is not any improvement as compared with the original solution.

1.5.2.2 Variation in data

We considered two different kind of variation in data in section 1.4.2.2: (1) Different data types; and (2) Semantically different data types.

For the first case, the problem is the same as for the the structural variability: methods already existing can not be overridden, therefore, joinpoint interception need to be combined with intertypes declaration. In this case, the joinpoint interception mechanism picks up those points where data conversions are required, and advice is executed, which calls to the an intertype declaration that contains the code for performing the data conversion. The placement of the data conversion code in a intertype declaration does not provide any extra benefit as compared with the original technique. The same problem appears for the conversion of semantically different data types.

1.5.2.3 Variation in behaviour

Three different kinds of variation in behaviour were identified in section 1.4.2.3. We comment on this kind of variabilities below.

Addition of new components with new functionalities

This kind of variation in behaviour requires mainly the addition of new classes (e.g. LightControllers) that support new functionalities. This implies that the previously existing (base) classes must incorporate references to these elements.

It was identified the in order to manage alternative references between classes, i.e. references that could be not present in a class since its multiplicity is zero, the base class requires to have this reference even when the number of instances to be created is zero. For instance, in the case of the Smart Home, the HouseGateway must have collections to each kind of device that could be
Figure 1.9: Intertype declaration solution for dealing with variation in behaviour

present in a specific SmartHome, independently of these devices belong to a selected feature or not. In case the corresponding feature were selected, the collection would be instantiated and values provided. Otherwise, the collection would remain empty.

This limitation can be overcome using *intertype declarations*. The alternative references (and all the code necessary to manage, such as getters, setters or other associated methods) can externally be added using an aspect with *intertype declarations*.

For instance, in the Smart Home case of study, and following the Figure 1.3. The HouseGateway class controls and has references to all the RoomControllers of the Smart Home. If automatic light management is added, a collection of LightControllers must be added to the class RoomController. This collection is externally added by means of intertype declarations.

Figure 1.9 depicts this solution. Line 01 defines the intertype declaration that adds an ArrayList of LightControllers to the class RoomController. Lines 02-03 contain the intertype declaration that adds the getLights method, for accessing to this the list of LightController objects, to the class RoomController.

**Variation in the implementation of a service**

Two different cases are considered: (1) variations that affect to only one service; (2) variations that affect to several services in the same interface or even several interfaces.

For the first situation, as already commented, the problem can be solved just using object-oriented techniques such as abstract classes, interfaces, inheritance and polymorphism (see section 1.4.2.3).

In the second situation, the use of intertype declarations, again, does not offer any improvement over the *joinpoint mechanism* for identical reasons than in previous section. Intertype declarations can not be used without joinpoint interception and the combination of both does not show any improvement over the original solution.

**Variation in how the different services are coordinated**

Again, intertype declarations can not be used without joinpoint interception and the combination of both does not show any improvement over the original solution.
Contributions  

*Intertype declarations* have been found helpful just for the case of extending classes with new methods and attributes that were not initially considered. As advantage, the extended classes does not need to be directly modified, and these extension can be kept separately of the extended class. As disadvantage, new conflicts could emerge due to conflictive intertype declarations, i.e. attributes and methods introduced by different aspects but with a same name and type or signature [HNBA07].

### 1.5.2.4 Variation in Quality

Variations in quality can be due to: (1) a variation in the internal implementation of one or more methods; (2) a variation in the quality attributes, e.g. security. Both cases can be managed as a kind of variability in behaviour described before, being the benefits and drawbacks of intertype declarations the same as in that case.

### 1.5.2.5 Variation in environment

Three different kinds of problems are identified: (1) distribution into different kinds of nodes; (2) different distributions for the software modules that are part of a specific product; and (3) dealing with different external services.

**Variation in the node type**

As already explained in section 1.4.2.5, aspect-orientation is helpful for managing references to alternative components, where each one of them is a specific version for a specific kind of node, e.g. PDA, mobile phone, laptop or common PC. The addition of intertype declarations to this solution does not improve it.

**Variation in the deployment configuration**

When the way the components are distributed among different nodes it also changes the way they communicate. A solution is to use aspects to capture the communication among the components, and in case they are in different nodes redirect the communication. We have no found any use for *intertype declarations* here.

**Variation in the external services used**

Changing an external service can be as viewed as managing alternative references, but, as commented in section 1.4.2.5, interfaces interfaces must match and no adaptation should be required. In case there were a mismatching between alternative interfaces, the problem can be solved by the use of aspect-oriented adapters. The incorporation of
intertype declarations to these aspect-oriented adapters has not been found helpful.

1.5.2.6 Variation in technology

Few contributions of aspect-orientation were found in section 1.4.2.6 for dealing with variability in technology. Unfortunately, the use of intertype declarations or the combination of intertype declarations with joinpoint interception, do not improve the situation.

1.5.3 Evaluation

A full evaluation of intertype declarations is not performed since not many differences with the joinpoint interception mechanism has been found and the use in isolation of intertype declarations does not seem to contribute to variability management.

Intertype declarations improve extensibility since it is possible to add optional members and methods to a class externally. This characteristic has been found useful for adding of new classes that provided new functionalities (e.g. light management). The benefit in both cases is that it is possible to add the new components and the alternative references even when the base classes were not previously prepared for that.

1.6 CaesarJ Feature-Oriented design

CaesarJ is an aspect-oriented language which unifies aspects, classes and packages in order to solve a set of different problems of both aspect-oriented and component-based programming [AGMO06]. The components are collaborations of classes, but they can modularise crosscutting features or non-functional concerns. CaesarJ is Java based and can be used in combination with plain Java.

This section describes how some of the mechanism introduced by CaesarJ could be used to manage variability in a SPL pointing out strengthens and weaknesses.

1.6.1 Description

The main contribution of CaesarJ over AspectJ is the introduction of a better type system by the use of advanced object-oriented modularisation mechanisms, such as “family classes”, “wrappers” and “mixing composition”.

A family class\(^8\) is a large-scale piece of functionality which involves a group of related classes. Abstraction, late binding, and subtype polymorphism is sup-

\(^8\) Other terms used in literature for family classes are collaborations, layers, teams or feature classes
ported at the level of family classes. A family class is a special CaesarJ class which can contain inner classes called "virtual classes". Just like methods and fields, they are also members of instances of their enclosing family class, called family object. Hence, at any time during execution their meaning is relative to the dynamic type of the family object. Subclasses of a family class can refine inherited inner classes (further-binding). In such further binding, we can override inherited methods, add new methods or new state, as well as add additional superinterfaces and superclasses.

Mixin composition is a composition mechanism that allows a family class to inherit from more than one super class, with the constraint that super classes must have a common super type. A special linearization method is used to avoid conflicts in the composition. Mixins were originally defined in Flavors [Moo86] as an alternative to multiple inheritance.

CeasarJ was designed aiming to be a language suitable for Feature Oriented Programming (FOP), which goal is to modularise software systems into features. Each feature should encapsulate all the functionality and dependencies between classes in a single module, which CaesarJ names family class. Features depend of other features creating a dependence tree which defines the structure of the SPL and helps to understand when and where is possible to add or modify features on the tree.

Such a feature decomposition should contribute to a better extensibility and reusability. This decomposition in features is of interest in a SPL context, in order to separate those features of a system that are variable from those ones that are common to a family of products [GA07]. Feature-oriented modularisation reduces the mismatch between specification and implementation, and provides a better support for independent development and testing.

The problem of feature separation is addressed by employing CaesarJ virtual classes and mixing composition. In CaesarJ each feature is modelled by a top-level class, while domain objects are modelled by their virtual classes. We will refer to such top-level classes as feature classes. Dependencies between features are modelled by inheritance relationship between the corresponding feature classes. In general a feature class inherits from another feature class in order to override its functionality. Since a feature may need functionality of multiple features is important to have some form of multiple-inheritance in the language. CaesarJ mixin composition is used for this purpose. An application is defined as a feature class that inherits from classes of all other features that must be available in the application.

Figure 1.10 shows a part of the feature-oriented design for the SmartHome case study, which will be used as example throughout this section. The structure of the SPL would be defined by the feature classes and the dependency graph. A product inside this SPL would be initially selected by creating a new feature class which inherits from all the leaf features of this graph. Some variants are chosen at this level by changing the features that are going to be used in the graph. After this, there is still necessary to instantiate it, in order to create a concrete instance of that product. It is at this moment when the final parameters for variability are provided, and the concrete product is totally configured and it is
Figure 1.10: Smart Home dependency diagram
ready to use. How each kind of variability can be properly solved using CaesarJ is going to be analysed in the next sections.

1.6.2 Kinds of variability

1.6.2.1 Variation in Structure

Using the terminology of section 1.4.2.1, for solving the “the problem of the variable constructor” (the creation of a variable number of instances of an object B associated with an object A), any change is required on the feature dependency graph, since the artifacts or components are already there, the unique element that varies is that there will be a different number of instances of such features in a specific product, e.g. a different number of rooms.

An instantiated object of virtual class A (e.g. RoomController), could contain a variable number of instances of a virtual class B, (e.g. LightController). The specific number of objects B to be created and associated with an object A can be defined as when the feature class that represent the final product is instantiated as in the “main” method, i.e. the entry point, of the application. This is similar to conventional object-oriented programming, and the same problems regarding variability management in structure appears here.

The example used in section 1.4.2.1 is refactored here to show a CaesarJ-based solution, which is not different to the solution employed in common object-orientation. The main difference is that we are instantiating now feature classes instead of common classes, but the mechanism is practically the same. Figure 1.11, line 00 specifies the definition of the final feature class. In this case, the graph, in this case the FRoom feature class is considered the leaf of the feature graph for simplicity. When the application starts, this feature is instantiated and a floor and room controllers are added to the product by means of the appropriate methods.

Excepting the difference due the use of Feature-Oriented approach, the solution presented in Figure 1.11 is the same as using common object-orientation. A particularity of CaesarJ is that is not possible to create instances of an inner classes of a feature class directly, so it is required the addition of some methods to the feature class, such as the addRoom method used in lines 05-06, that have the responsibility of creating these instances of inner classes adequately.

In the case of the “behaviour structurally dependant” (see section 1.4.2.1),
using only CaesarJ feature classes, it is necessary to adding a new feature class
to the feature dependence graph. This new feature inherits from the features
involved in the structurally dependant behaviour. This behaviour requires being
added just in case a specific objects has been instantiated. For instance, using
the same example as in section 1.4.2.1, the LightController objects must notify
to the WallScreen object each time its intensity is modified if and only if there is
a WallScreen object instantiated for the room where the light is.

The new feature class redefines those methods of the inner classes that re-
quire being extended with a new behaviour (e.g. LightController). Each redef-
inition must check first if the corresponding instances that requires the extra
behaviour exist (e.g. a WallScreen for the same room of the light), and if so, the
extra behaviour (e.g. a notification to the WallScreen) is then executed.

Using the example of the LightControllers and the WallScreen, a new feature
NotifyWallScreen is created. This feature redefines the methods of the Light-
Controller virtual class that modify the light intensity. If there were a WallScreen
instantiated for the same room of an light whose value is adjusted, a call the
adjustLight method of the WallScreen object is performed.

This solution is illustrated in Figure 1.12. The new feature class FNotify-
WallScreen inherits from FLightActuator since it needs to redefines some inner
classes of FLightActuator (line 00). The class LightController is redefined (line
01) in order to redefine those must be extended with the new behaviour. The
addIntensity method is then redefined (lines 02-08).

Contributions

The solution for dealing with “the problem of the variable con-
structor” is the same as busing object-orientation, excepting some slight differ-
ces due mainly to the use of feature classes instead of common classes.
There is not any special benefit on using CaesarJ for this kind of variability
as compared to traditional techniques. On the other hand, the use of feature
classes add an small extra complexity, as the developer needs with some par-
ticularities of these constructions, such as inner classes can not be created
directly from static methods.

The solution for solving the “the problem of behaviour structurally dependant”,
using just feature classes without jointpoint interception, is also very similar to
the object-orientated one. Nevertheless, the use of feature classes provides

```java
00 public cclass FNotifyWallScreen extends FLightActuator & FWallScreen{
01   public cclass LightController{
02       public void addIntensity(int value){
03           super.addIntensity(value);
04           WallScreenController controller;
05           controller=getWallController(floorNumber,roomNumber,lNumber);
06           if (controller!=null){
07               controller.adjustLight(lNumber,intensity);
08           }
09       }
10   }
```

Figure 1.12: CaesarJ solution for dealing with structural variability (II)
some benefits because all the inner classes that must be modified are encapsulated inside the same feature class. When an inner class is modified by feature class inheritance the changes are visible for the rest of the classes without requiring castings or specifying which inner class is going to be used. Therefore, changes in the code already created are not required, improving reusability, extensibility and evolution.

If several methods must be modified using the same piece of code, this solution is not the best option since this piece of code is scattered across several methods. Feature classes without joinpoint interception do not offer a solution to solve adequately this problem. Thus, it is recommended to combine feature classes with joinpoint interception for this case.

1.6.2.2 Variation in data

Two different kinds of variations were variation in data were identified in section 1.4.2.2: (1) different data types; and (2) semantically different data types. In the first case, if artifacts are modelled as feature classes is possible to create a new feature class which inherits from one of the artifacts and to redefine the corresponding methods in order to adapt them to the new data types. Variability is archived by adding or not the new feature class at compile time. As explained before, the final SPL is constructed by creating a new feature class that inherits from all the selected features of the dependency graph. Thus, if the feature class specifying the variation in data is added, this variation is incorporated to the product. Otherwise, it is not. In this case, polymorphism and inheritance are used in the same way as an object-oriented solution. Thus, there is not a big difference between both solutions. If a data conversion is not feasible using just object-orientation; the conversion will not also be feasible using CaesarJ.

The variation could affect to several methods of the one or several classes. If all the inner classes are in the same hierarchical path, all methods redefinitions can be carried out in a a single feature class, keeping the dependencies encapsulated in that feature.

For instance in the Smart Home, as already commented, some conversions with data related to temperatures might be required. Using a CeasarJ-based solution, the HeaterTemperatureSensor would be modelled as a feature class, instead of defining a new class like in object-orientation (see Figure 1.13, line 00). For dealing with the variation in data, a new feature class HeatingSensorFahrenheit is created, which inherits from HeatingTemperatureSensor and redefines the method getExternalTemperature (see Figure 1.13, lines 02-03) in order to adapt the data types.

Contributions

The presented solutions use inheritance and polymorphism in order to achieve variability in data. The solution is quite similar to the object-oriented solution,
Figure 1.13: CaesarJ solution for dealing with variation in data (I)

```
00    public cclass FHeatingSensorFahrenheit extends 
       FHeatingTemperatureSensor{
01       public cclass HeatingController{
02           public int getExternalTemperature () {
03               return convertToFahrenheit(externalTemperature);
04           }
05           ... 
06       }
```

Figure 1.14: CaesarJ solution for dealing with variation in data (II)

having the same advantages and drawbacks. Some benefits are obtained since feature classes helps to hide dependencies between classes as these dependencies are encapsulated in a single modularity unit, the feature class. All the required modifications to deal with the variation in data conversion are performed inside the same feature class. As previously commented, changes performed inside a feature class are visible for the rest of the classes being not necessary any modification in the already existing code, which contributes to reusability, extensibility and evolution.

The selection between variants is as simple as including a feature or not in the final feature class.

1.6.2.3 Variation in behaviour

Three different kinds of variation in behaviour were identified in section 1.4.2.3. We comment on them below.

Addition of new components with new functionalities

This kind of variation in behaviour requires mainly the addition of new classes (e.g. LightController), which supports the new functionalities. This implies that the previously existing (base) classes must manage references to these elements and new classes, with new dependencies between them, must be added to the existing code. Feature classes of CaesarJ helps to this process, as all the logic related to the new feature is encapsulated into a single module, a feature class, that is added to the dependency graph. This feature class inherits (or extends) as many already feature classes as required. Including this feature or not in the a specific product can be achieved by simply adding this feature class to the feature class that represent the specific product. Without using feature
classes, all the classes contained in the feature class should be added individually, i.e. one by one, to the final product, which is more tedious, error-prone and time consuming. Additionally, the application engineer must be aware of the dependencies between this inner classes. Furthermore, management of dependencies between feature classes is also alleviated since the selection of a feature class implies automatically the selection of all the feature classes that the former feature class requires.

**Variation in the implementation of a service** In this case, we distinguish between two different kinds of variations: (1) variations that affect only to one service; (2) variations that affect to several services in the same interface or even several interfaces.

For the first situation, the problem can be solved just using object-oriented techniques. However, some benefits, already commented, regarding encapsulation of dependencies, can be obtained from the use of CaesarJ.

In the second situation, using CaesarJ, all the methods that need to be redefined can be redefined into a single feature class, but each redefinition has to be done separately for each method. The problem can not be solved just using feature classes, and we recommend to combine this mechanism with joinpoint interception due to the crosscutting nature of the problem.

For instance, in the SmartHome case of study, all methods of the LightController class that changes the value of the light intensity have to be redefined in order to send a notification to the GUI controller. These methods belongs to the class LightController of the feature class LightActuator (see Figures 1.15 and 1.16). A new feature class LightGUI is created. This feature class inherits from the feature classes LightActuator and from GUI. The method setIntensity of the class LightController is properly redefined in the feature class LightActuator. Lines 04-05 show the crosscutting code that has to be added to all the methods to advise the GUI of the modifications, in this case we are printing the new intensity value in the GUI.

**Variation in how the different services are coordinated** In this case, the protocol that governs the interaction between different software entities changes. The crosscutting nature of this problem has already been discussed in section 1.4.2.3. Due to this crosscutting nature, a solution using only feature classes
is not feasible, being required its combination with joinpoint interception.

**Contributions**

In the case of “addition of new components with new functionalities”, interesting benefits are obtained from the use of feature classes. As already commented, feature classes acts as an encapsulation unit that improves variability management (less code need to be created/generated, management of dependencies is automatically solved in some cases).

In the case of “variation in the implementation of a service”, no special benefits has been observed as compared with object-orientation and/or joinpoint interception.

In general, feature classes provide some benefits since variants can be encapsulated into single units, keeping the dependencies between inner classes well-modularised inside a feature. As already commented, this improves reusability, extensibility and evolution.

**1.6.2.4 Variation in quality**

Variations in quality can be due to different reasons, as already explained in section 1.4.2.4: (1) a variation in the internal implementation of one or more methods; (2) a variation in the quality attributes.

Both cases can be managed as a kind of variability in behaviour described before. In the first case, polymorphism and inheritance like in object-oriented solution could be used. In the second case, the crosscutting nature of the quality attributes could not be directly solved using just feature classes. Therefore a combined use with joinpoint interception is suggested.

**1.6.2.5 Variation in environment**

We distinguish three different kinds of variation in environment:
1. The software modules that comprise a specific product are distributed into
different kind of nodes.

2. The software modules that are part of a specific product could be dis-
tributed on nodes differently.

3. A specific product deals with different external services.

We comment on each one of them below.

Variation in the node type  In this case, different versions of a same compo-
nent exist depending on the kind of node where the component it is going to be
deployed. The problem has two different faces: (1) implementing several com-
ponents that share functionality and they are similar with the minimum effort and
the better maintenance, evolution and reusability as possible; and (2) managing
alternative references to these changing components.

The first case, the problem the required changes are not crosscutting and they
are very well localized. It could be solved using object-orientation by inheritance
and overriding of methods.

In the second case, it is necessary to manage alternative references to the
changing components. Using CaesarJ, a solution is to define each alternative
component in a different feature class. The feature class contains all the inner
classes necessaries to define the component, keeping the dependencies be-
tween classes isolated and encapsulated. The new feature has to inherit from
all the features it depends on to have access to them. The feature class con-
structor will be implemented to add the alternative reference when the final fea-
ture class is instantiated. In order to do that, the alternative components have to
implement a common interface or inherit from the same abstract feature class.
It is even possible to add a new alternative references that were not presented
before by adding members to the feature or inner classes using inheritance, and
instantiate it to the desired component in the constructor.

For instance, in the Smart Home case of study, two different kinds of light
devices can be acquired. Depending on the devices,a different software com-
ponent must be deployed in them. The solution provided by CaesarJ would be
to replace the feature class LightManagement by an abstract feature class in
the dependency graph (see Figures 1.17 and 1.18). This abstract feature class
has a reference to a LightManagement object, which it is not still instantiate. The
feature classes LightManagement and AdvancedLightManagement are created,
both inheriting from the abstract feature class. Each subfeature class instanti-
ates a different version of the LightManagement object, realizing the variation. In
the abstract feature class AbstractLightManagement is defined extending from
the two feature classes it has dependencies with. A partial definition of the final
feature class that defines the application is shown in Figure 1.18, line 11.

Variation in the deployment configuration  When the way the components
are distributed among different nodes, it also changes the way these compo-
nents communicate, as requirements for remote communications can emerge
or disappear. Using feature classes, a solution is to inherit from the feature that implements the component and redefine the methods that must deal with remote communication, relying on some kind of middleware technology (e.g. Java RMI). The new feature will be selected statically in the final feature class when the component is in a different node.

This solution has two main drawbacks: (1) the code needed to communicate with the middleware is crosscut among several methods and therefore it has to be added manually for each method to be redefined by the feature class; and (2) if the redefined feature class is not a leaf of the dependency tree, the features that are lower in the hierarchy could not see the changes done by the new feature, so the original dependency tree need to be modified in order to do these feature classes inherit from the newly created one. It is also possible to create abstract feature classes, where the clients of these features depend only on the abstract definition, while the final product will use the client with specific implementations of the abstract feature classes.

**Variation in the external services used** Changing an external service can be as easy as changing a reference, when the interfaces of the alternatives match and no adaptation is required. Techniques for changing references using CaesarJ feature classes has already been commented.

In case the interfaces, protocols or both do not mismatch, there we have not found a straightforward solution using CaesarJ feature classes. A potential
solution would be to create a feature class that inherits from all the features were the external service is used and redefine all the methods involve in the communication with the external service in order to ensure the compatibility between interfaces.

For instance, in the Smart Home case study, the SmartHeatingManagement feature could have the functionality to automatically send a request to the gas provider for refilling the gas tank when this is getting empty. The protocol to access this service varies between different gas providers. A feature class that inherits from the feature SmartHeatingManagement would be created. This feature would redefine the methods which calls to the gas provider in order to adapt them to the current protocol interface.

In addition, CaesarJ offers the possibility of using *wrappers* for adapting classes to a certain interface, preserving interesting properties such as polymorphism. Thus, CaesarJ wrappers are an interesting option for dealing with this kind of variation. Nevertheless, we consider wrappers as a kind of aspect with improvements regarding typing, having the same benefits and drawbacks as already commented for joinpoint interception.

### Contributions

CaesarJ provides benefits for managing alternative versions of a component where each version is associated to a specific kind of node (e.g. PC, PDA or mobile phone). The use of feature classes help to manage dependencies between classes encapsulated inside a feature class, with the same benefits regarding extensibility, reusability and evolution previously commented. Nevertheless, the solution is quite similar to common object-orientation, sharing more or less the same benefits and drawbacks.

When the variation is related to the deployment configuration or variation of the external services that an application uses, CaesarJ feature classes alone are not enough, being recommended to use feature classes in combination with joinpoint interception. In the latter case, variation of the external services, CaesarJ wrappers improves the solution offered by AspectJ due to the better type system of CaesarJ.

#### 1.6.2.6 Variation in technology

We have identified two potential sources for variation in technology at the implementation level (see section 1.4.2.6): (1) a change the programming language; (2) change the set of abstracts, primitives, techniques or guidelines that are used to construct the software.

In the first case, there are few work that can be done using feature classes, as usually the only possible solution is to have the system replicated once by each target language.

In the second case, it is not possible to give a general solution, as already commented (see section 1.4.2.6).
CaesarJ feature classes method is suitable when it is necessary to introduce modifications or even new references that were not presented before in already existing classes. This could also be done using just object-orientation, but feature classes allows developers to avoid modifications on the already existing code, which is not possible using only object-orientation. In case of crosscutting appeared, the combination of feature classes with joinpoint interception is recommended.

For instance, if the communication protocol must be changed from IPX to TCP/IP, and in case the communication is modelled inside a single feature class, a new feature class would be created, which inherits from the previous one. This feature class could even add and initialized socket references in several inner classes of the program if it would be required.

1.6.3 Evaluation

This section presents a general evaluation of the feature classes mechanism, offered by CaesarJ, as a mechanism for implementing variations.

Expressive Power

Variation by inheritance between feature classes is a interesting mechanism for management of variability in a SPL context, mainly due to the improvements regarding reusability and the possibility of extending the SPL with new features in an easy way, which contributes to scalability and evolution. Since the dependencies between related classes are encapsulated inside features, the addition of features to the hierarchical tree is often enough for realizing the variability.

Feature classes has revealed to be suitable for variations in data, behaviour, quality and environment variability, excepting when problems related to crosscutting concerns appear. In case of crosscutting problems appeared, it is recommended to combine feature classes with joinpoint interception.

The selection of variants is managed at compile time including or not leaf features of the dependency graph into the feature class that represents a specific product. In case of variation in structure, it would be necessary to add the initialization code to the constructor of the feature class that represents the final product. This could be automated by the use of code generators.

CaesarJ can be considered as an incremental variation mechanism, since new functionality is added over already existing features. As many variants as desired can be selected at the same time, whenever there were not violations of the dependencies between features.

However, the manual creation of all the dependency graph for a given feature model could be a repetitive, tedious and time consuming task. The problem can be alleviated by means of automating the generation the CaesarJ skeletons for
the dependency graph from a feature model using model-to-code transforma-
tions.

**Binding Model**

Variants are often bound at compile time. CaesarJ also support dynamic as-
pect deployment so it could be combined with Feature-Oriented Programming to
allow a dynamic selection of variants during execution. However, since CaesarJ
uses invasive weaving, the inclusion off all the possible features in the compiled
code to support full dynamic variability could imply a big overhead.

**Validation**

The availability of variation type in most cases is inferred by inheritance of
feature classes. In cases like structural variability it is explicitly specified in the
main method by or in the constructor or the final feature class. The specification
of the variation type is not very precise when polymorphism is used for the
variability. The reason is that the relation between feature classes and how
to use the functionality of other feature could not be very clear and it varies
depending of how each feature of the graph is defined.

A complete validation may be impossible. Since dependencies between fea-
tures could not be defined using CaesarJ and the Feature-Design implementa-
tion described, we are not able to specify all constraints that are necessary
to specify which selection of features will result on valid products. However,
a partial validation can be achieved, since the SPL could be constructed in-
crementally following the dependency graph and in each step the variability in-
duced by inheritance could be validated. The problem appears when a full
feature is changed, in these cases, since the dependencies are not fully spec-
ified and there are not explicit constraints, it is difficult to ensure the impact of
the changes. However, this problem could be overcame if the SPL is designed
by the use of abstract feature classes.

**Modularity**

Feature-Oriented programming with CaesarJ makes dependencies explicit in
the feature dependency graph. These dependencies are described at the level
of feature classes and not at the level of virtual classes, whose dependencies
are well-modularised inside a single feature class. It is also possible to express
decomposable classes by declaring them as virtual classes. Feature classes
are not decomposable, but they are used as modules to decompose the ele-
ments inside them. Methods can be decomposed too, since it is possible to
have multiple overridings of a single method. The overriding methods will be
composed during mixin composition of the enclosing feature classes.
It is not completely possible to specify dependencies in CaesarJ just using interfaces, because there is not any differentiation between the features implementing an interface (abstract feature) and those using it.

The dependency of reusable code in the variation could be mainly settled in two categories. Stable abstraction when the variability is solved using inheritance and therefore polymorphism between feature classes. Unaware abstraction is also achieved in structural variability in which the variant is specified by instantiating the classes in the main method of the application. The decomposition of reusable assets could be done in different feature models thanks to the mixing composition provided by CaesarJ.

**Reuse**

The different features of the SPL could be combined in order to create different specific products of the same SPL with more or less functionality. Inheritance between features is allowed and therefore, could be and reused by other features to increase the functionality or add new artifacts. The features achieve a good nice reusability inside the same SPL since dependencies between features are explicit but in this kind of design it is not easy to reuse them in a different project.

**Extensibility**

The SPL is constructed starting from a root feature and developing a hierarchical dependency graph. The addition of new features to an already created SPL is as easy as adding at any moment a new feature to this graph that extends all the upper features (an some of their methods) in the hierarchical tree. From experience, these extensions often requires few or even no changes in the already existing modules.

**Size**

Experiments and benchmarks reveals an overhead on final lines of code after compilation of approximately a 25%. This increase is mainly caused by the finer granularity of the decomposition into classes and methods in the Feature-Oriented design.

**Complexity**

The number of classes and virtual classes generated by Feature-Oriented Programming is high in comparison with the same problem solved using object-orientation. But if the complexity is analyzed under the scope of dependencies the scenery is completely the opposite. In object-oriented design, most
of classes are interdependent causing cyclic dependencies. In the feature-oriented design, this problem is reduced because dependencies between features are strictly acyclic, which means that if a feature A depends on a feature B, we can assert there is not a dependency in the opposite direction, either directly or indirectly. The acyclic design is in fact enforced, because we model feature dependencies by class inheritance relation [GA07].

**Understandability**

The high level design has a clear structure, which corresponds to the structure of the feature model. The inner classes of a feature class contain only the attributes and methods that are relevant to this feature. In order to understand how the feature works we need to study only the feature and the features it depends upon. We can even combine these features into a working application and experiment with their behaviour in isolation from other features. However, it is not very intuitive that feature classes can use the functionality of transitively inherited features [GA07].

**Testability**

The advantage of the CaesarJ design for testing is that features can be tested independently from other unrelated features. For each feature group is possible to define a minimal application that consists only on the features of this group and their dependencies and used this application to test the features. In object-oriented applications test cases are often organized by use cases, which by their nature are similar to features, but such test cases usually depend upon much more source code than they actually need to test the corresponding functionality [GA07].

**Maintainability**

The improved traceability of requirements in the implementation also helped for bug fixing, because it was easier to locate the cause of the bug. A big advantage for maintainability is also the clear acyclic dependencies between features, because it is easy to identify what features can be evolved independently from each other and what is the potential impact of the change in a certain feature [GA07].

**Tool Support**

CaesarJ has a very limited tool support at the moment: integration of CaesarJ compiler and source code edition in Eclipse platform, outline and inheritance hierarchy views, as well as rudimentary support for debugging. The modular type
checking provided by compiler and the debugging support are essential for enabling Feature-Oriented Programming with CaesarJ. However, for efficient programming there is also a strong need for navigation and search in the structure of the source code, code completion and automatic generator imports. Another problem is that CaesarJ compiler did not support incremental compilation, so any small change requires the full recompilation of the project.

**Efficiency**

This technique should not imply any performance overhead, as the weaving performed by the CaesarJ compiler is static and invasive, the woven code should be as efficient, as in memory as in time, as the normal object-oriented code. As it was shown in the size section the amount of code generated is bigger so in this aspect a small lack of performance could be detected.

1.7 HyperJ: multidimensional separation of concerns

This section discuss on the applicability, benefits and shortcomings of the multidimensional separation of concerns approach to variability realisation and variability management in the scope of a Software Product Line.

1.7.1 Description

*Separation of concerns* refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose [OT00]. Each different concern as a class or a feature is referred as a dimension of concern. Separation of concerns involves decomposition of software according to one or more dimension of concerns.

Developers must be able to identify, encapsulate, modularise, and manipulate multiple dimensions of concern simultaneously, and to introduce new concerns and dimensions at any point during the software lifecycle, without suffering the effects of invasive modification and rearchitecture.

Modern languages and methodologies, suffer from the problem *tyranny of the dominant decomposition* [TOHJ99], because they permit the separation and encapsulation of only one kind of concern at a time. The term multi-dimensional separation of concerns is used to denote separation of concerns involving:

- Multiple, arbitrary dimensions of concern.
- Separation along these dimensions simultaneously.
- The ability to handle new concerns, and new dimensions of concern, dynamically, as they arise throughout the software lifecycle.
Overlapping and interacting concerns; it is appealing to think of many concerns as independent or “orthogonal,” but they rarely are in practice.

Multidimensional concerns are managed inside a hyperspace. A hyperspace is a concern space specially structured to support multi-dimensional separation of concerns and permit the explicit identification of any concerns of importance, encapsulation of those concerns, identification and management of relationships among those concerns, and integration of concerns.

A syntactic construct in a programming language is named an unit, inside the hyperspace approach. The units in a hyperspace are organized in a multi-dimensional matrix, in which each axis represents a dimension of concern, and each point on an axis a concern in that dimension. This makes explicit all the dimensions of interest, the concerns that belong to each dimension, and which concerns are affected by which units. The coordinates of a unit indicate all the concerns it affects; each unit affects exactly one concern in each dimension. Each dimension can thus be viewed as a partition of the set of units: a particular software decomposition. Any single concern within some dimension defines a hyperplane that contains all the units affecting that concern.

Hyperslices are sets of units that are declaratively complete, which means they must declare everything to which they refer encapsulating the dependencies between all the units it contains.

Declarative completeness is important because it eliminates coupling between hyperslices. Instead of one hyperslice referring to another, thereby depending upon the other specific hyperslice, each hyperslice states what it needs by means of the abstract declarations, thereby remaining self-contained. Units, concerns and hyperslices do not exist in isolation; they can be interrelated by one or more integration relationships that indicate how they are to be combined.

HyperJ [TOHJ99] is an approach to multidimensional separation of concerns in Java, by the use of hyperslices as declaratively complete set of Java units.

Regarding the variability realisation and management in the context of a Software Product Line, the solution will be encapsulated each variant into a hyperslice and a specific product would be obtained by composition of hyperslices, using the adequate set of integration relationships. As hyperslices are declarative complete, feature reusability, extensibility and evolution would be increased, since coupling between features is reduced. Hyperslices can be considered as a symmetric approach, since two combined hyperslices are equally relevant, and any of them is an extension of the other one. In the case of CaesarJ, feature classes are asymmetric, since, according to the dependency graph of CaesarJ, feature classes are combined by means of inheritance, which is an asymmetric relationship.

Nevertheless, the symmetric nature of HyperJ does not provide special benefits in the context of Software Product Lines, which are mainly asymmetric, defining a core of base assets that are extended by the addition of some variants over that core. One benefit of hyperslices as compared with feature classes could be reusability of features across different SPL, as hyperslices are not dependent of any core or base assets. However, this kind of reusability is not of-
ten required, such as most of features are usually specific of each SPL and not reusable across different projects. For those features, that are not dependent of a specific project or software product line, the use of interface between class collaborations or feature classes can help to improve reusability on CaesarJ, and adequately combined with joinpoint interception, is most of times enough. Therefore, CaesarJ presents a good degree of multidimensional separation of concerns, being hyperslices not strictly required. Additionally, the complexity of the HyperJ language, its lack of effective tool-support and the instability of the compiler are other reasons for considering the use of CaesarJ instead of HyperJ for developing Software Product Lines.

### 1.8 Dynamic weaving: the DAOP platform

CAM/DAOP [PFT05] is a component and aspect based model and platform that applies the principle of separation of concerns to separate both components and aspects as first class entities. In this platform these entities are then dynamically woven at runtime using a non-invasive weaving strategy.

In this section we will describe briefly the CAM/DAOP platform. Conceptually, as variation mechanism, there is not conceptual differences with the previously evaluated technologies. For this reason, CAM/DAOP is not evaluated against each kind of variability as before, as the results will be the same. The particularities of the weaving mechanism of the DAOP platform provides some extra benefits, as compared with the other techniques, in the context of a SPL. In addition, CAM/DAOP also presents some benefits regarding management of dependencies between features for component-based software development [Szy02]. We discuss on both issues in this section.

#### 1.8.1 Description

The main elements of CAM/DAOP are:

1. **CAM (Component-Aspect Model) [PFT05]**, a new model to design component and aspect based application specified in UML. The CAM model defines the main entities of a CAM application and the relationships among them. Therefore, aspects are identified at the design level contrary to other aspect proposals, where the separation of aspects is delayed until the implementation level.

2. **DAOP-ADL [PFT03]**, an XML-based Architecture Description Language (ADL) that is used to describe the structure of CAM applications in terms of a set of components, aspects and composition rules.

3. **DAOP (Dynamic Aspect-Oriented Platform) [PFT05]**, which is a distributed component and aspect based platform. DAOP implements the CAM model and provides a dynamic composition mechanism that plugs software aspects into components at runtime.
Another relevant feature of this approach is how the information provided with the DAOP-ADL language is used at runtime by the DAOP platform:

1. The information about the application architecture is loaded in the internal structures of the DAOP platform when the application is initiated;

2. The DAOP platform consults this information to obtain the composition rules that indicate how to perform the dynamic composition of components and aspects.

1.8.2 Variability dependencies management

As commented in the previous sections, Aspect-Oriented Programming (AOP) helps to solve the problem of crosscutting variables features and dependencies between features. Nevertheless, Aspect-Oriented Software Development (AOSD) promotes the separation of concerns at every stage of the software lifecycle, from requirements and architectural design (early aspects) to implementation. Therefore, instead of generating an implementation in an aspect oriented programming language directly from a feature model, SPL in CAM/DAOP are developed specifying first the non-crosscutting and crosscutting features at the architectural level using DAOP-ADL.

The benefits of doing so are, that the software architect can manage the evolution of crosscutting features and dependencies at architectural level; it is possible to reason and solve crosscutting feature dependencies before accomplishing the implementation task, which is less time consuming and error prone and, since the aspect-oriented architecture specification eradicates the crosscutting feature and dependencies problem, the implementation generated from it will be much simpler than for example considering a direct implementation in AspectJ [FG07].

In the Feature Oriented Analysis (FOA) [LK04] approach a feature may intuitively correspond with an architectural component (although this is not always so). But there are some features that crosscut other parts of the product line architecture, which drastically reduce reusability, adaptability and the evolution of one product line architecture. This problem is still worse for variable features of a family product, since they may affect other components of the product line architecture. As a consequence, all the components influenced by a variable feature have to be modified in accordance with this variation (cf. the problem of the behaviour structurally dependent in section 1.4.2.1). If a crosscutting variable feature is encapsulated in an aspect, different architectural configurations can be specified without modifying other components.

An additional problem is that if variable features are dependent on each other, their variation may produce chained changes affecting different base elements of the product line architecture. This problem is called invasive change. To solve this problem, the dependencies between features must be analyzed and separated from the other components.

Let et al [LK04] identify six kinds of dependencies:
• The Usage dependency means that a feature depends on other features for its correct functioning.

• The Modification dependency occurs when the behavior of a feature may be modified by other feature.

• Exclusive-Activation dependency happens when different features must not be active at the same time.

• Subordinate-Activation dependency occurs when a feature can be active only if another specific feature is also active.

• Concurrent-Activation dependency means that some subordinators of a superior (in a Subordinate-Activation dependency) must be active concurrently while the superior is active.

• Sequential-Activation dependency denotes that some subordinators of a superior must be active in sequence.

Aspects can be used for encapsulating and separating these dependencies. Thus, the impact of changes in a chain of variable features can be reduced.

As said before, AOSD promotes the separation of concerns at every stage of the lifecycle. In the context of SPLs, the commonalities might be decoupled from variabilities by means of managing common features as base modular components and some of the variable features as aspectual components, which adapt or extend the base modular components. In addition, common features that crosscut other parts of the architecture can also be modelled as aspects. The effects of the variations of crosscutting variables features can be minimized by modelling such features in separate entities as aspectual components. In this way, this variable feature can be incorporated into the product line as a positive variability without modifying any other components of the product line architecture. Variable features that do not crosscut other components can be simply designed as common components.

In component-based systems, dependencies among features have to be taken into account at architectural level in order to avoid the invasive changes problem. These dependencies must be analyzed and separated from other components at the architectural level, when the components of the application are defined. Each kind of dependency can be treated to be encapsulated in an aspectual component to facilitate the localization of the effects of a feature variation (See Table 1.1).

<table>
<thead>
<tr>
<th>FOA dependency</th>
<th>DAOP-ADL</th>
<th>Explanation</th>
</tr>
</thead>
</table>

V1.0 2007-12-07  AMPLE WP3 D3.2  58 of 146
### Usage:
Common component makes use of a variable component.

Model the common component as DAOP-ADL component. Model the variable component as a DAOP-ADL aspect.

Reversing dependencies: variable aspectual components make use of the common component.

### Modification:
Common component being modified depends on variable component.

Model the common component as DAOP-ADL component. Model the variable component as a DAOP-ADL aspect.

Reversing dependencies: the modifications are managed in the variable aspectual component instead of in the common component.

### Activations:

| Activations dependencies. | A DAOP-ADL Property stores whether a feature is active or not. | That property must be checked in the guard of the variable aspectual component to activate it or not. |

**Table 1.1: Mapping between FOA dependencies and DAOP-ADL**

The problem with the *Usage dependency* can be formulated at architectural level where common component makes use of a variable component. Then, in products where the variable component is not required, the common component has to be modified.

Something similar happens with the *Modification dependency*. The problem occurs when the common component being modified depends on a variable component. Such dependencies are solved by reversing dependencies, i.e. by modelling the variable component as an aspect; the dependency from common components to variable component is reversed to a dependency from the variable component (aspectual component) to the common component.

Others dependencies can be grouped as Activation dependencies. This kind of dependency is solved using the properties of DAOP-ADL (see Table 1.1).

Therefore, using aspects and properties we solve the dependencies between features without affecting other components, as is shown in Table 1.1. This table describes how the dependencies between features can be solved with architectural elements in DAOP-ADL.

We illustrate these concepts using an excerpt of the Smart Home case study. Figure 1.19 depicts the feature model corresponding to the Smart Home basic facilities.

Regarding the dependencies, the *Authentication* feature must be active only while the *AutomaticOpen* feature is active. Then, they have a Subordinate-Activation dependency. Another dependency exist between the *Authentication* feature and the *Fire Management* feature. Because when a fire is detected the...
**Figure 1.19:** An excerpt of the feature model of the Smart Home case study

Fire Management must be active but the Authentication has to be deactivated in order to avoid that the firemen have to be authenticated (which can be really problematic). Then Fire Management and Authentication features must not be active simultaneously. This is an Exclusive-Activation dependency.

**Description of the Smart Home in Non-Aspect Oriented Component Architectures** Figure 1.20 shows the diagram for a non aspect-oriented component-based architecture corresponding to the excerpt of the Smart Home shown in Figure 1.19. Dashed boxes indicate optional components (optional features). Dashed ovals represent the interface that depends on optional components. The connectors has been omitted at this point for the sake of simplicity.

If AutomaticDoor and AuthenticationDoor are selected, some components have to be modified since some interfaces must be added. Specifically, the DoorManagement component has to be modified. The AutomaticDoor component has some messages in its provided interface to open-close doors automatically. Therefore, A required interface (AutomaticRI) must be added to the DoorManagement component. The AuthenticationDoor component has different messages in its provided interface for authenticating people crossing doors. Then, in the same way, a required interface (AuthenticationRI) must be added to the DoorManagement component for binding this component with the AuthenticationDoorComponent.

This example demonstrates that some components have to be changed depending on the selected features. The selection of the optional component AutomaticDoor and AuthenticationDoor requires changes in the DoorManagement component.

The second problem under analysis is the dependency between features. In this example we have two dependencies: (1) a Subordinate-Activation between Authentication and Automatic features; and (2) an Exclusive-Activation between Authentication and FireManagement features.
For the first dependency the AuthenticationDoor component requires a message to know if AutomaticDoor is active, and for the second dependency the AuthenticationDoor need a message from FireManagement component in order to know if it is active or not. Then, those components have to provide and require those messages. The AutomaticDoor component must add to its provided interface a message called isAutomatic that indicates if the automatic feature is selected or not. The AuthenticationDoor must incorporate a required interface with target AutomaticDoor to require the isAutomatic message. The same issue appears with the FireManagement component, it must add to its provided interface a message called fire that indicates if the FireManagement is active and if there is fire. Then, the AuthenticationDoor that has to incorporate a required interface with target FireManagement to require the fire message. Again, some components have to be modified to solve the dependencies.

Partial Smart Home in DAOP-ADL. In order to show the advantages of using aspects, the previous example is refactored with aspects, illustrating how components affected by crosscutting variable features do not have to be changed. The crosscutting variable features are modelled as aspects. In the previous example, there were two Activation dependencies that are now solved using properties.

Figure 1.21 shows the diagram for the Smart Home example with components, aspects and aspectual connectors. There are four optional aspectual components and one aspectual connector.

In this case, the architectural components are not modified depending on if AutomaticDoor and AuthenticationDoor are selected or not.

The DoorManagement has the same event interface (DoorEI), that had when
AutomaticDoor and AuthenticationDoor were not activated. AutomaticDoor and AuthenticationDoor have evaluated interfaces with joinpoint ANY, i.e. any pointcut that is specified in the AspectualConnector.

The AspectualConnector has to capture some events thrown by DoorManagement. After these events are sent, the AutomaticDoor or AuthenticationDoor aspectual components have to be executed.

Using this approach, to solve the Subordinate-Activation dependency and the Exclusive-Activation we can use the properties of DAOP-ADL [PFT03]. These properties are used to encapsulate shared data that has to be accessed from different components and aspects.

There will be two boolean properties called isAutomatic and thereIsFire, which are true if the door is automatic and there is fire respectively and false in any other case. These properties will be consulted in the AuthenticationDoor guard. Then, the AuthenticationDoor will be active when authentication feature is selected, the door is automatic and there is no fire.

Again, we solve the dependencies between features using aspects and without modifying other parts of the architecture. The crosscutting variables features and the dependencies are encapsulated in aspects and in the aspectual connector. In addition, component interfaces of both common and variable components are much simpler than in a non-AO approach.

1.8.3 Non-invasive dynamic weaving

The DAOP platform performs dynamic and non-invasive weaving, which brings interesting benefits for the development of Software Product Lines.
DAOP composes, i.e. weaves, component and aspects dynamically at run-time by interpreting the information contained in DAOP-ADL, thus, bridging the gap between architecture and implementation. This weaving mechanism offers the possibility of weave and unweave aspects at runtime (components can also be loaded and unloaded at runtime), which allows features can be selected and unselected at runtime. Furthermore, as these components and aspects can be independently developed and compiled, it is possible to develop new components and aspects while the application is running and dynamically add these components and aspects to the application, without requiring stopping and re-compiling the system. Therefore, features can be selected dynamically at runtime, resulting on reconfigurations of the applications. These kind of dynamic reconfigurations are quite common in ubiquitous and/or pervasive computing or open systems [GTLJ07, RSP05].

In addition, the weaving performed by the DAOP platform in non-invasive, which means entities crosscut by aspects are not modified as a result of the weaving process. This kind of weaving is mandatory in component-based software development [Szy02], where software applications are develop by assembling prefabricated, or off-the-shelf components, that are managed as black-box entities and are often available only on binary form. Similar problems appears in Service-Oriented Architectures (SOA), where services has also a black-box nature and can not be internally modified during a weaving process; and legacy systems, which are long-used and tested systems that, for business constraints, should be only updated or modified if and only if it were strictly necessary.

1.8.3.1 Evaluation

As commented before, the variation mechanisms are conceptually equivalent to joinpoint interception, so a complete evaluation against all the points of the evaluation criteria is not shown here, as it will show similar results to obtained in section 1.4. We comment only in those parts where differ. DAOP improves traditional joinpoint interception techniques, such as AspectJ [KHH+01] regarding variants binding time. Variants can be bound and unbound at runtime. Furthermore, variants can be really unbound in the sense that when an aspect or component is unload, it is unload from the memory, i.e. variants are not longer part of the product. DAOP also improves variant availability, since new variants can be added to the application after it is compiled and while it is running.

As negative point, the DAOP platform have an overhead in runtime performance due to the dynamic composition process. However, DAOP was designed for developing highly distributed web applications, where the networks latency and delays make this overhead performance not noticeable. When this overhead were a limitation, JAsCo [SVJ03] could be a suitable alternative. JAsCo offers an optimized dynamic and non-invasive weaving process, with similar characteristics to the DAOP platform, but with a runtime performance similar to static weavers. On the other hand, dynamic weaving helps to reduce the size in memory of the product as variants unselected at runtime can be really removed from the product, no occupying space in memory.
Component-based architectural representations are useful to variability management in Software Product Line, as they offer high-level representations of the product that eases variant selection and configuration.

One disadvantage of using DAOP-ADL and CAM/DAOP to develop product lines is that this platform is not made specifically to develop this kind of software. Then, if we want express that certain components or aspects are optional or variables in the architecture we will need to extend DAOP-ADL adding these possibilities. This task is not very difficult because DAOP-ADL is XML-based and we can extend it adding some tags and guards. In [FG07] is detailed this extension and how we can solve the dependencies between features.

Regarding tool support, the CAM/DAOP approach provides an IDE for that assist engineers in the development of aspect-oriented component-based applications [FPT07].

This IDE is implemented as a set of Eclipse plug-ins\(^9\).

Using this IDE, the DAOP-ADL model of the application can be loaded into an Architecture Description and Validation tool (Figure 1.22), by which software architects can easily consult and share the description of the software architecture with the different stakeholders. Optionally, this tool can serialise this architectural information to make it available to the DAOP platform.

Moreover, in order to facilitate the reuse of components and aspects, the CAM/DAOP IDE also contains a Component and Aspect Directory tool, which provides access to a web-based repository of components and aspects.

---

\(^9\)It can be downloaded from http://caosd.lcc.uma.es/CAM-DAOP/tools
1.9 JET code generation

Java Emitter Template (JET) has been developed to simplify the process of automatic code generation (Java, XML, JSP, etc) using template code generators. JET is target language agnostic and therefore, it is possible to generate any kind of code using JET.

JET technology shares some similarities with the well-known technologies that generates dynamic web server pages, such as Java Sever Pages (JSP) or PHP, since templates are translated to a Java class to be lately executed by a generator class created by the user. JET could be combined with Eclipse creating plug-ins with graphical user interfaces which could simplify the process of providing parameters for these templates.

The collection of templates which implements an application is called the blueprint. The parameters that serves for instantiating the template are referred to as the input model. These parameters are provided by means of an XML file. A blueprint should define a schema describing what parameters it is expecting. The JET engine will lately read the blueprint and the parameters file and combine them in the final source code (see Figure 1.24).
public class HouseGateway {
    ...
    public static void main (String [] args){
        HouseGateway hg=new HouseGateway();
        <c:iterate select="/structure/room" var="currRoom">  
            RAux=new RoomController("<c:get select="$currRoom/@id" />",
                <c:get select="$currRoom/@number" />);
            <c:iterate select="$currRoom/light" var="currLight">  
                RAux.add(new Light("<c:get select="$currLight/@id" />",
                        <c:get select="$currLight/@number" />");
            </c:iterate>                    
            hg.addRoom(RAux);
        </c:iterate>               
    }}}}  

Figure 1.25: Jet solution for variation in structure (I)

The template includes fixed code that is written directly in the syntax of the target language. This code can be decorated with special tags that serves to control the presence or not in the output of the template depending on the value of the parameters of the input model. Multiple output files can be generated using JET (see Figure 1.24).

This section explores how JET can be used for realizing and managing variability in the context of a Software Product Line. Different versions of a variable artifact are developed at once by means of the creation of templates that contains all the versions of this artifact. Providing parameters to this template, a specific version of this artifact is automatically generated. Code of any programming language (or other textual artifacts) can be generated using JET. Nevertheless, for consistency with the previous sections and for the sake of readability, we will continue using Java.

1.9.1 Kinds of variability

1.9.1.1 Variation in structure

The problem of “the problem of the variable constructor” (see section 1.4.2.1) can be reduced using JET. In this case, the main problem is that a large amount of code need to be created by each product inside of a SPL depending on how many instances of several features (e.g. lights, heaters) are selected. The problem can be alleviated is a configuration file is created and parse for the application when this is initialized. Nevertheless, this configuration file need still to be created and usually involves a large amount of information.

JET can be used to parameterize this artifacts and automatically generate as the initialization code as the previously commented configuration files based on the information of the input model instead of having to specify all this amount of information manually.

For instance, in the SmartHome case of study, the HouseGateway has as-
associated a variable number of RoomControllers and each RoomController is associated with a variable number of LightControllers. The main method inside HouseGateway is parameterized to create a different number of objects when the application is initialized depending on the contents of the parameters input (see Figure 1.25). In Figure 1.25, line 04 iterates on the different rooms on the input model (see Figure 1.26). A new RoomController is created by each room of the put model, with the id and number specified in the input model. Line 07 iterates on the different lights for the current room. Line 08-09 creates a LightController object by each light. After this, the RoomController created is added to the HouseGateway.

The "problem of the behaviour structurally dependant" (see section 1.4.2.1) can not be directly solved using template code generators. The reason is that this problem implies to deal with variable instances of one class, at not with variable classes, which is what Jet allows us to express. For instance, using the example of previous sections, LightControllers must sent notification messages to WallScreens if and only if there is a WallScreen in the same room where the light is. This implies that the instances of the class LightController will have a variable behaviour depending on the room where they are deployed. A feasible solution would be to create two classes LightControllers, one with notifications and the other one without them. One class would inherit from the other one. Then, at the initialization of the product, e.g. main method, the template code generator will create an instance of one of these two classes depending on if there is or not a WallScreen in the same room where the light is created.

Contributions Jet enables the parametrisation of initialization code (e.g. main methods, constructors or configuration files) in charge of creating variable number of instances in different products. The objects to be created, their relationships their initialization parameters are obtained from an input model and the initialisation code automatically generated. Without Jet, this code has to be created manually, which is a laborious, time consuming and error-prone task.

User friendly interfaces, such as graphical interfaces or wizards can be created to generated the XML files that contains the parameters of the input model.

However, the parameterised initialisation artifacts (e.g. main methods) can be hard to to read and understand when the number of parameterised elements and code generator blocks is high. Type checking, testing and debug-
ging is more complex since there is not any real and usable (compilable and executable) code until the template is instantiated. To check that the template is correct, or at least compile, for each potential instantiation would imply to generate all the possible configurations of the templates. In a typical software product line, the number of potential configurations is usually really high\textsuperscript{10}, being almost impossible to generate all of them in order to ensure, at least, they compile.

JET could be combined with joinpoint interception in order to avoid the necessity of specify manually the body of the advices, when joinpoint interception is applied to deal with variation in structure (cf. section 1.4.2.1). Using JET, advices bodies can be parameterized and automatically generated by each specific product. The same idea applies to CaesarJ and Feature-Oriented Programming.

JET is not suitable to solve in a straightforward way “the problem of behaviour structurally dependant”. It can combined with object-orientation. In that case, Jet helps to select which specific class of an class hierarchy must be selected depending on the variants selected.

1.9.1.2 Variation in data

Two different problems regarding variation in data are considered: (1) different data types; and (2) semantically different data types.

In the latter case, a possible solution is to parameterise the methods which deal with the variable data in order to do the required data conversion depending on the variants selection, which is obtained from the input model.

For instance, in the SmartHome case study, the class RoomHeatingController communicates with the class TemperatureSensorController. Both classes might work either Celsius or Fahrenheit. The sensor works by default in Celsius. The method getTemperature is then parameterized in order to convert the temperature value to Fahrenheit, before returning, just in case this variant is selected (see Figure 1.27). In Figure 1.27, line 03 the parameter tempUnit of the RoomController of the input model is selected to test which unit is being used. If the value is “C” (line 04), it means the RoomController is working in Celsius and therefore no conversion is required. If the value is “F” (line 05) the temperature value is then converted to Fahrenheit before returning it.

\textsuperscript{10}For the Smart Home case study described in this document, the fmp plugin calculates a total of 3.145.728 configurations.
As before, other possible solution would be to use inheritance for creating
two different classes, each one working in a different measurement units. Jet
would help to select which specific class of an class hierarchy must be selected
depending on the variants selected.

These concepts are illustrated using the previous example (see Figure 1.28). The TemperatureSensorController is instantiated in the RoomHeatingController
constructor. A second TemperatureSensorController that inherits from the the
previous one and redefines the method getTemperature is created. The con-
structor of the heating controller class is parameterized to instantiate adequately
the temperature sensor to the correct class.

The solution for the second case does not differ from the previous one, and it
has the same advantages and drawbacks.

**Contributions** Data conversions can be achieved by parameterising object meth-
ods. Variants can be selected just changing the input model. If the same conver-
sion would to be performed in several methods, a scattered and tangled piece
of template code generator would be required in each class demanding data
conversions. In this case, a feasible solution would to be to encapsulate the
crosscutting piece of code in aspects with parameterise pointcuts. According to
the information of the input model, this pointcut will select certain methods of
certain classes.

Regarding extensibility, the addition of new variant implies the manual modifi-
cation of the templates.

In the second case, combination of JET plus inheritance and polymorphism,
the product is easier test and debug, since the different class representing vari-
ants are not parameterized, they can be compiled and tested as normal object-
oriented code. In case of crosscutting appeared, joinpoint interception would be
again required.

**1.9.1.3 Variation in behaviour**

Three different kinds of variation in behaviour has been identified. We comment
on them below.

**Addition of new components with new functionalities** The first kind of vari-
ation in behaviour requires mainly the addition of new classes (e.g. LightCon-
troller), which supports the new functionalities. This implies that the previously existing (base) classes will require managing references to these elements. The solution would be to parameterise the classes with optimal references. For instance, in the SmartHome case study, the HouseGateway will have a collection of LightController objects if and only if automatic light management is selected. Thus, this attribute and the associated methods for manipulating this data structure (e.g. getters and setters) are surrounded by Jet tags that will include these pieces of code just in case automatic light management were selected. The build file has also to be parameterised in order to include or exclude certain classes of the compilation unit according with the selection of variants.

Variation in the implementation of a service In this case, we distinguish another two different kinds of variations in the implementation of a service: (1) variations that affect to only one service; (2) variations that affect to several services in the same interface or even several interfaces.

For the first situation, variation in the implementation of a service, the problem can be solved just using object-oriented techniques such as abstract classes, interfaces, inheritance and polymorphism, as happened in the previous sections. Jet can help to select which specific class of an class hierarchy must be selected depending on the variants selected.

In the second situation, there would be a crosscutting piece of code. A solution with JET is to parameterise the methods which implementation varies depending of the selected variant, and select the correct implementation depending of the parameters of the input model, but it does not solve the problems associated with scattered and tangled representations. Again, the combination of Jet with joinpoint interception is required.

Variation in how the different services are coordinated In this case, the protocol that governs the interaction between different software entities changes. Coordination protocols are crosscutting pieces of code. Jet allows the parametrisation of the code related to coordination, and thus, we can switch between different coordination protocols just changing the parameters of the input model. Nevertheless, there is not solution to the crosscutting problem, being the use of joinpoint interception recommended.

Contributions Solutions employed are similar to previous sections, thus, benefits and drawbacks are the already commented ones. In general, Jet simplifies the selection between variants, as only changes in an input model are required. However, it is not possible to solve problems associated with crosscutting concerns.

1.9.1.4 Variation in quality

Variations in quality can be due to different reasons: (1) variations in the internal implementation of one or more methods of one or more classes; (2) variations
in the quality attributes of an application. Both cases can be managed as a kind of variability in behaviour described before.

In the second case, due to the crosscutting nature of the quality attributes, a Jet solution is not recommended since the required parametrisation would lead to scattered and tangled template code blocks. Joinpoint interception is the recommended option.

Templates could be also used under the variation in quality scope to deal with fine grained optimizations of code, which is a typical scenario for using macros and conditional compilation in C/C++.

1.9.1.5 Variation in environment

Three different kinds of variations in environment are identified:

1. The software modules that comprise a specific product are distributed into different kind of nodes.

2. The software modules that are part of a specific product could be distributed on nodes differently.

3. A specific product deals with different external services.

We comment on them below.

Variation in the node type In this case, the family model is most likely to have different versions of the same component depending on the kind of node where this component it is going to be deployed. The problem has two different faces: (1) implementing several components that share functionality and they are similar with the minimum effort and the better maintenance, evolution and reusability as possible; and (2) managing alternative references to this changing components.

In the first case, changes are well localized and do not crosscut the SPL artifacts. It can be solved just using inheritance and overriding of methods to create the different classes. Jet could be used to simplify the process of selecting which concrete class need to instantiate depending on the variants selection, as in previous sections. The second case is similar and it can be solved using similar mechanisms.

Variation in the deployment configuration When the way the components are distributed among different nodes it also changes the way in which they communicate, since components that were previously in a same node could be now in different nodes and they could require some middleware technology (e.g Java RMI) for communicating remotely.

A solution is to parameterize all the methods that need remote communication and redirect the communication to the middleware, in case remote communication were required.
When two classes A (e.g. HomeGateway) and B (e.g. HouseDatabase) of a SPL could be moved between different nodes, they are parameterised, placing the code related to remote communications between Jet alternative tags. Depending on the variant selection, which is reflected in the input model parameters, the classes will communicate through the middleware or not.

Other potential solution would be to inherit from the class A, and redefine all the methods that communicates with the B in order to use the middleware. As before, Jet would help with the selection of the right concrete class to instantiate. However, if more than two components can be moved between nodes increases, the number of concrete classes increases highly, being this option not recommended.

Variation in the external services used Changing an external service can be as easy as changing a reference, when the interfaces of the alternatives match and no adaptation is required. Techniques for changing references using JET has already been commented in previous sections.

If a mismatch between interfaces appeared, Jet can be use to parameterize the methods that requires adaptation.

For instance, in the Smart Home case study, the SmartHeatingManagement class could be implemented as a template and the methods which calls to the gas provider parameterized in order to adapt them to the interface of selected gas provider service. However, this imply that the adaptation need to be performed at compile time, when this kind of adaptation are usually required at runtime, allowing the user to switch between different services according to some criteria (e.g. to use the cheapest gas provider, the fastest or a middle solution between prices and delivery time).

Contributions As before, some problems can be solved just using inheritance and polymorphism and Jet can help to select which specific class of an class hierarchy must be selected depending on the variants selected.

In the case of variation on how components are deployed on nodes, Jet does not offer any improvement over previous techniques. For variations in the selection of external services, the JET solution is not optimal as variants need to be selected at compile time, and, in this cases, the selection between variants is made at runtime in most of cases.

1.9.1.6 Variation in technology

Two potential sources for variation in technology at the implementation level: (1) a change the programming language; (2) change the set of abstractions, primitives, techniques or guidelines that are used to construct the software.

Regarding changes in the programming language, JET can be used to generate code in any programming language. The solutions and examples have been presented in Java but it could be written in any programming language. Nevertheless, if we want to generate two products in two different languages, we
need still to deal with two sets of template code generators, one for each programming language. One advantage in this case is that the same input model can be used to configured products written in different languages. Other advantage is that it is possible, inside a same product, to have different parts of same product written in different programming languages (e.g. some components in Java and some components in C). Changes on modules written in different languages can be controlled by parametrisation, since Jet is unaware of what it is generating. This crosslanguage variability management is not possible with aspect-oriented techniques, which are mostly language-dependent.

1.9.2 Evaluation

This section contains a general evaluation of template code generators as a mechanism for dealing with variability in the context of SPL.

**Expressive Power** Template code generators allows the creation of a template piece of code that can be customised according to certain parameters. This technique is mainly based on parametrisation of the source code. Variants are not separated of base assets; instead they are hard-coded on them.

This technique can be mainly used to solve problems related to variation in structure. It is also helpful for selecting the right classes to instantiate in class hierarchies, according to some parameters provided as input.

**Binding Model** Variants are bound to base assets before compiling, at the time of generating the source code. Therefore, variants are mainly static using Jet. In addition, variants are also only available at generation-time but not a run-time.

Some techniques could be combined with Jet in order to allow some kind of feature selection at runtime, such as, for instance, the strategy pattern. Let us suppose a system with five possible variants in an algorithm. The strategy pattern is used to select at runtime one of the possible variants. However, not all variants could be available at runtime. The infrastructure code for supporting the strategy pattern can be parameterized by means of a template code generator. Then, before compiling time, the code generator could produce the code for three of the five strategies, so two variants would not be available at runtime and the other three would be.

**Validation** One of the main problems of template code generators is that type checking can not be performed before generating the source code. This can be problematic for large templates with a large amount of parameters and possible configurations, since in order to check the whole template is valid, all the possible configurations need to be generated and compile, which for a medium scale project could be completely unrealistic. As commented before, for the
Smart Home case study, using the fmp plugin, 3,145,728 configurations has been calculated.

**Modularity** The level of separation of concerns is the same one we find in object-orientation so there is not any special mechanism for separation of cross-cutting concerns using JET. The granularity of separation is at method level or class level, depending on if the variability is solved by parameterisations of methods or parameterisations of initialization of variables.

In case of dependency of reusable code on variation, JET is in the category of inline variation but using a more advanced system than C/C++ conditional compilation.

There is not any special mechanism for decomposition of reusable assets using JET, since it depends on how the SPL is implemented.

**Extensibility** JET uses templates to generate variable source code; however the most of the SPL code is plane object-oriented code. Extend the SPL requires modifications in the plane object-oriented code or in the templates, in both cases at implementation level these modifications are invasive and have to be done manually. Since object-orientation is used, the number of classes or templates that could require modifications, and the associated effort, would depend on the specific extension to perform and how the SPL is implemented. The use of parametrisation does not provide special benefits here since normally an extension requires introducing new parameters in the templates.

**Size** The size of the generated code is the same one than codifying the selected variants manually. In the case of the templates, it depends on each product. Parameterisations involves more lines of code, as the generator statements need to be included as an addition to the normal source code. But, since one code generator can help to save multiple lines of repetitive lines of code (e.g. iterations in the template), it is not possible to draw a general conclusion of the size of the source code using template as compared to hand-made code.

**Complexity** The complexity of the generated code is the same one as of one application manually created, since JET only instantiates the templates to plain source code. In the case of the templates code, there is an increment of complexity due to extra information coming from input model needs to be included in order to perform the parameterisations.

**Understandability** Templates themselves are usually hard to understand, mainly because the developer needs to perform the instantiation process mentally in order to reason about what is really happening in the code, as well as being unaware of the actual semantics of the parameters of the input model. The most the code is parameterized the most complex is to understand it in general. In
the generate code, there should not be any lack of understandability since it is normal code.

**Testability** Template code generator does not offer any improvement or drawback on testability of the generated product as compared with normal techniques, such as object-orientation. Templates themselves, however, can not be directly tested. Any test requires waiting until the final source code is generated and this final source code depends of the parameters input. Therefore it is not possible to do a general testing which ensures the correct functionality of any final product. Even worse, it is also not possible to ensure in an easy way that a template compiles for all its possible configurations.

**Maintainability** Template code generators does not offer any maintainability advantage over traditional techniques, such as object-oriented programming, since the maintainability of the final code depends on the the mechanisms of the generated code. Maintainability at template level becomes hard, since the final code depends of the parameters input. It is not easy to find errors or dependencies before having the final source code generated. Modifications in the templates to correct bugs or doing improvements depends on how they are implemented but in any case is harder to understand and do modifications in a template than over plane object-oriented code.

**Tool Support** JET has an integration eclipse plug-in to facilitate the generation of code, but due to the nature of template code generators, it does not have support for debugging. A desirable characteristic will be to been able to prevvisalise the code that will be generated by the JET engine, but this option is not available at the moment. JET could be included in eclipse plug-ins that support the selection of variants by means of user-friendly graphical editors or wizards.

**Efficiency** JET engine generates normal source code, equivalent to the code that would be generated manually. Therefore, there should not be performance overhead in time or memory since the code generated is the same as if it were manually created.

### 1.10 Conclusions

This section summarizes the results of the previous section, outlining benefits and drawbacks of each variation mechanisms as compares to the others. The next table resumes the main results of the analysis and evaluation carried out throughout this first part of this report. Each column refers to one of the aspect-oriented or model-driven variation mechanism analysed. Each row refers to a particular variability scenario (e.g. changing the coordination protocol) inside a specific kind of variation (e.g. variation in structure). Each cell contains a
short sentence that outlines if the corresponding variation mechanism improves or not the current state-of-art and a brief justification for that sentence. If there is improvement, the variation mechanism is considered as suitable for dealing with that kind of variation, if there is not such an improvement, it is considered as not suitable. For the cases where the mechanism can be used but without convincing benefits, the variation mechanism is considered just as usable.

This table contains only the three AOP variation mechanism evaluated and JET. DAOP is not shown as the advantages of using a dynamic platform as conceptually the mechanisms of CAM/DAOP are equivalent to joinpoint interception, but with some technical improvements due to the dynamic non-invasive weaving. Hyperslices are also not shown as it was concluded they do not offer any special improvement, in the context of Software Product Lines, as compared with feature classes of CaesarJ.

<table>
<thead>
<tr>
<th>Variation in structure</th>
<th>Joinpoint interceptions</th>
<th>Intertype declarations</th>
<th>Feature classes</th>
<th>Template Code Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>The problem of the variable constructor</td>
<td>Not suitable (the separation does not provide clear benefits)</td>
<td>Not suitable (not different from joinpoint int.)</td>
<td>Not suitable (similar to OO)</td>
<td>Suitable, (automatic generation of initialization code)</td>
</tr>
<tr>
<td>The problem of behaviour structurally dependent</td>
<td>Suitable (especially in crosscutting cases, separates the variation)</td>
<td>Not suitable (not different from joinpoint int.)</td>
<td>Suitable (in absence of crosscutting, improves extensibility and evolution)</td>
<td>Usable (in absence of crosscutting, helps with variant selection)</td>
</tr>
</tbody>
</table>

**Table 1.2: Comparison regarding variation in structure**

<table>
<thead>
<tr>
<th>Variation in data</th>
<th>Joinpoint interceptions</th>
<th>Intertype declarations</th>
<th>Feature classes</th>
<th>Template Code Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different data types</td>
<td>Usable (avoid the use of castings, quantification helps to deal with crosscutting)</td>
<td>Not suitable (no improvement on joinpoint int.)</td>
<td>Suitable (improves extensibility and evolution)</td>
<td>Suitable (in absence of crosscutting)</td>
</tr>
<tr>
<td>Semantically different data types</td>
<td>Suitable (especially in crosscutting cases, allows separation of variants)</td>
<td>Not suitable (no improvement over joinpoint int.)</td>
<td>Suitable (improves extensibility and evolution)</td>
<td>Suitable (in absence of crosscutting)</td>
</tr>
</tbody>
</table>
### Table 1.3: Comparison regarding variation in data

<table>
<thead>
<tr>
<th>Variation in behaviour</th>
<th>Joinpoint interceptions</th>
<th>Intertype declarations</th>
<th>Feature classes</th>
<th>Template Code Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition of new compo-</td>
<td>Not suitable (not clear</td>
<td>Suitable (enables the</td>
<td>Suitable</td>
<td>Suitable (automatically</td>
</tr>
<tr>
<td>nents with new</td>
<td>mechanisms for adding</td>
<td>extension of base assets</td>
<td>(eases variability</td>
<td>generation of the in-</td>
</tr>
<tr>
<td>functionalities)</td>
<td>new interfaces)</td>
<td>with new interfaces/</td>
<td>management)</td>
<td>itialization code and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>attributes)</td>
<td></td>
<td>addition of new</td>
</tr>
<tr>
<td>Variation in the</td>
<td>Not suitable (no</td>
<td>Not suitable (no</td>
<td>Not</td>
<td>Usable (helps with</td>
</tr>
<tr>
<td>implementation of one</td>
<td>improvements on OO)</td>
<td>improvement on</td>
<td>suitable</td>
<td>variant selection)</td>
</tr>
<tr>
<td>service</td>
<td></td>
<td>joinpoint int.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variation that</td>
<td>Suitable (due to the</td>
<td>Not suitable (no</td>
<td>Not</td>
<td>Not suitable (not solve</td>
</tr>
<tr>
<td>affects several</td>
<td>crosscutting)</td>
<td>improvement on joinpoint</td>
<td>suitable</td>
<td>crosscutting)</td>
</tr>
<tr>
<td>services or interfaces</td>
<td></td>
<td>int.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variation in how the</td>
<td>Suitable (decouple</td>
<td>Not suitable (no</td>
<td>Not</td>
<td>No suitable (no</td>
</tr>
<tr>
<td>different services are</td>
<td>coordination from</td>
<td>improvement on</td>
<td>suitable</td>
<td>mechanism to decouple</td>
</tr>
<tr>
<td>coordinated</td>
<td>computation and helps</td>
<td>joinpoint int.)</td>
<td></td>
<td>coordination from</td>
</tr>
<tr>
<td></td>
<td>to modularise</td>
<td></td>
<td></td>
<td>computation)</td>
</tr>
</tbody>
</table>

### Table 1.4: Comparison regarding variation in behaviour

<table>
<thead>
<tr>
<th>Variation in quality attribute</th>
<th>Joinpoint interceptions</th>
<th>Intertype declarations</th>
<th>Feature classes</th>
<th>Template Code Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method implementation</td>
<td>Not suitable (no</td>
<td>Not suitable (no</td>
<td>Suitable</td>
<td>Suitable (helps with</td>
</tr>
<tr>
<td></td>
<td>improvements on OO)</td>
<td>improvement on</td>
<td>(improves</td>
<td>variant selection)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>joinpoint int.)</td>
<td>extensibility</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>and evolution</td>
<td></td>
</tr>
</tbody>
</table>
Quality attributes

<table>
<thead>
<tr>
<th>Suitable (as quality attributes are most of times crosscutting)</th>
<th>Usable (can help to extend interfaces if required by join-point int.)</th>
<th>No suitable (not solve crosscutting)</th>
<th>No suitable (not solve crosscutting)</th>
</tr>
</thead>
</table>

Table 1.5: Comparison regarding variation in quality

Variation in environment

<table>
<thead>
<tr>
<th>Joinpoint interceptions</th>
<th>Intertype declarations</th>
<th>Feature classes</th>
<th>Template Code Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components dependent on node type</td>
<td>Not suitable (there are not clear benefits)</td>
<td>Usable (can help to extend interfaces)</td>
<td>Suitable (improves extensibility and evolution)</td>
</tr>
<tr>
<td>Deployment configuration</td>
<td>Usable (helps to encapsulate remote communication code)</td>
<td>Not suitable (no improvement on joinpoint int.)</td>
<td>No suitable (not solve crosscutting)</td>
</tr>
<tr>
<td>External services</td>
<td>Suitable (eases the implementation of external adapters)</td>
<td>Not suitable (no improvement on joinpoint int.)</td>
<td>Usable (in absence of crosscutting)</td>
</tr>
</tbody>
</table>

Table 1.6: Comparison regarding variation in environment

Variation in technology

<table>
<thead>
<tr>
<th>Joinpoint interceptions</th>
<th>Intertype declarations</th>
<th>Feature classes</th>
<th>Template Code Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming language</td>
<td>Not suitable (no solution)</td>
<td>Not suitable (no solution)</td>
<td>Not suitable (no solution)</td>
</tr>
<tr>
<td>Set of abstractions, primitives and guidelines</td>
<td>Usable (highly dependent on each case)</td>
<td>Not suitable (no solution)</td>
<td>Usable (highly dependent on each case)</td>
</tr>
</tbody>
</table>

Table 1.7: Comparison regarding variation in environment

The following conclusions can be derived from the previous table:

1. Intertype declarations (e.g. introductions) do not seem to provide any benefit from the variability management point of view as compared with joinpoint interceptions in the most of the cases, excepting with some base assets need to be extended with new attributes and interfaces. In most is
cases, they require being combined with joinpoint interceptions.

2. Joinpoint interception has demonstrated to be a interesting mechanism for dealing with certain kind of variability problems, particularly when dealing with these problems implies dealing with crosscutting pieces of code. This is not surprising, since joinpoint interception was precisely created for solving the lack of modularisation of crosscutting concerns (variable or not). From the 14 kinds of variability studied, 6 of them could be solved with a improvement as compared with the current state-of-art using joinpoint interceptions. Other important benefit of this mechanism is that is able to modify a previously existing code without any manual modification on it (although depending on the language, access to the source code could be required in order to perform the weaving). This characteristic makes joinpoint interception particularly suitable for implementing component adapters and coordinators.

3. Feature classes provide some benefits related to reusability, evolution and scalability, since dependencies between variants are well-encapsulated. The main variation mechanism behind feature classes is inheritance and polymorphism. The encapsulation of dependencies in feature classes alleviates the problems related to variability management.

4. JET offers the best solution, from the analysed ones, for dealing with the problem of the variable constructor, since it allows the generation of large amounts of initialization code in an easy and manageable way.

5. In any aspect-oriented case, variability management depends on if a certain aspect is introduced or not into a compilation unit. These mean a specific make or ant file need to written for each specific product inside a SPL. Jet can be used to construct a template build file, which generates a specific build file for a given configuration of variants.

6. When encapsulating variants into aspects, some parts of these aspects need to be manually codified at the application engineering level. Typical cases are pointcuts for aspects encapsulating quality attributes, such as, for instance, encryption. Depending on the selection of variants, the pointcut will select more or less methods. These parts often follow a well-defined pattern than can be specified using a template code generator that, with the adequate input parameters, generates the corresponding part of the aspect (e.g. the pointcut) for a specific product and variants configuration, avoiding the manual codifying of that part.

Finally, we would like to point out that no one of the analysed mechanisms is able to solve all kinds of variations. Each one of them offers different advantages and disadvantages for each particular case. Therefore, it can be conclude that an adequate variability management should focus on the combination of all of them, using the most suitable one in each particular scenario. For instance, in
the case of addition of new components with new functionalities, CaesarJ contributes to make the extension of component interfaces and variability management easier, but it can be still complemented with Jet for improving the selection of specific feature class in a concrete final product.
2 Extensions and Improvements to Implementation Technologies

2.1 Introduction

In this chapter we describe the state-of-the-art technology of the partners and propose a set of extensions to address the problems identified in our previous deliverable D3.1 [PRG+07] and the evaluation of the previous chapter.

In the deliverable D3.1 [PRG+07] we identified a set of evaluation criteria for support of variability in software product lines and used these criteria to evaluate various variation mechanisms. One of the conclusions of this evaluation is that different variation techniques have different advantages and disadvantages and none of them can replace all the others. The parametric mechanisms are very expressive and are unreplaceable for dealing with complicated fine-grained variability, but they can only deal well with preplanned variability and are usually difficult to extend with new variants. On the other hand, the refinement mechanisms provide a lot of flexibility for extension and are especially useful for dealing with unanticipated variability, but they fail when the variations are too complicated and fine-grained. Other typical trade-offs of variation mechanisms are, for instance, the trade-off between extensibility and modularity (in the sense of information hiding) and the trade-off between expressivity and possibility to validate various useful properties, such as type safety.

One way to improve existing technology would be to alleviate the disadvantages of a particular variation mechanism while preserving its advantages. Such improvements are however possible up to a certain degree, because the trade-offs are often of fundamental nature, which means that from a certain point improvements with respect to one design goal would inevitably require compromises with respect to other criteria. For instance, if we consider the trade-off between expressivity and validation it is well-known that validation certain properties, such as termination, are not decidable in a Turing-complete language. So a more modest goal would be to design variability mechanisms that achieve a better balance with respect to the identified trade-offs.

In D3.1 we identified various modularity issues as the biggest problems of AOP. The separation of the aspects from the modules that they advise is only at the structural level. It is difficult to understand the dependencies between aspects and the base code. There are no explicit interfaces between the base code and the aspects that would describe contracts between them and would enable safe evolution of the base code. Independent development of aspects is also problematic, because interaction of independently developed aspects can
lead to unexpected effects or even to conflicts in case of intertype declarations. These problems were also identified in the concrete scenarios of the previous chapter.

Modularity of aspects is one of the main arguments of AOP sceptics and one of the major obstacles for acceptance of AOP in industry. Therefore, we devote a lot of attention to this problem in our proposals. In Sec. 2.2 we explain how the modularity problems of aspects could be alleviated by virtual classes and propagating mixin composition. Further, we propose solutions to various remaining issues, such as differentiation between providers and users of interfaces (Sec. 2.3.1), hiding internal dependencies from clients (Sec. 2.3.2) and providing control over extensions from the side of the clients (Sec. 2.3.4).

In the survey of variation mechanisms of D3.1 we also identified the problem of transition between static and dynamic variation. While dynamic mechanisms can also deal with static variation, it is not possible to express dynamic variation by static mechanisms. It is a problem when a static mechanism very well suits to our specific requirements for variation management, but it cannot be used, because variation must be bound at runtime. Thus the existing technology can be improved by introducing more dynamicity to the existing variation mechanism. A typical example of such improvement is dynamic aspect deployment in CAESARJ [AGMO06, MO03], which makes it possible to vary crosscutting functionality at runtime. However, propagating mixin composition, which is another mechanism of CAESARJ used for modularization of product line features, [GA07] is completely static. In Sec. 2.3.6 we propose an extension that makes the mixin composition available at runtime. Besides, in Sec. 2.3.8 we address the problem of supporting domain specific languages (DSL) at runtime.

Another issue identified in D3.1 is lack of high-quality large scale parametric mechanisms. A flexible large-scale parameterization is provided by template languages, which are very popular in the context of MDD. However, we have identified that templates have bad support for validation and modularity, because the program code inside templates is not checked and the dependency links from and to this code are not made explicit. Furthermore, templates do not support incremental compilation. The trade-off between the flexibility of parameterization and possibility to check various properties is fundamental and thus these problems cannot be completely removed. In Sec. 2.3.7 we address this issue by our proposal of dependent classes, which is a more constrained, but a type-safe alternative for parameterization of classes.

We also address the problem of extension and composition of aspects. While aspects provide a lot of flexibility for extending the existing source code, the aspects themselves are not very extensible. In particular they lack constructs for extending and composing definitions of pointcuts. A proposal for extensible pointcuts is described in Sec. 2.3.3.

Since the trade-offs with respect to various criteria identified in D3.1 cannot be completely removed, different variation mechanisms would be the most suitable for addressing specific variation scenarios. This means that a good technology should provide a selection of different mechanisms that address different variability problems. Thus, another way to improve state of the art is to develop a
coherent technology, which provides a set of variation mechanisms that complement each other in order to cover a broad range of requirements to variation management.

In the previous chapter we evaluated various AOP and MDD approaches and identified the ways how they can complement each other. We propose various ways to integrate AOP and MDD. First of all, we propose to use an aspect-oriented language as a target for code generation. This will be done in cooperation with other work packages as described in Sec. 2.4. Furthermore, we propose to extend the MDD technologies with the elements of AOP. In Sec. 2.3.10 we introduce a joinpoint model and a weaver for models. Then we also propose to use aspects for expressing positive variability in model transformations (Sec. 2.3.12) and in code generation templates (Sec. 2.3.11).

Furthermore, in this chapter we also consider integration of various AOP approaches. CAESARJ, a programming language developed by TUD, already integrates various variation mechanisms in one programming language. In particular it integrates the advantages of classes, aspects and packages. Nevertheless, CAESARJ lacks a lot of features that are available in the technologies of EMN: quantification over sequences of events that is enabled by the stateful aspects of EAOP or dedicated support for concurrency and distribution that is available in respectively CEAOP and AWED. These technologies are described in Sec. 2.2. In Sec. 2.3.5 we describe integration of EAOP and CAESARJ, and in Sec. 2.3.8 we propose modular extensions of a programming language with support for concurrency and distribution. These extensions are based on modelling crosscutting behavioral and architectural concerns through domain specific languages (DSLs), which can also be seen as an interesting combination of MDD and AOP technologies.

2.2 Context

This section gives an overview of the existing technology of the partners that will be used as a basis for the extensions and improvements proposed in the next section.

2.2.1 CaesarJ

CAESARJ is an aspect-oriented language developed by TUD, which unifies aspects, classes and packages in order to solve a set of different problems of both aspect-oriented and component-based programming. Unification of packages and classes is achieved by supporting virtual classes and propagating mixin composition. Unification of aspects and classes means that the pointcuts and advice can be defined directly in the classes. As a result the aspects can be dynamically instantiated and deployed. They can be referenced by variables and used in a polymorphic way. As any other CAESARJ classes aspects can also have virtual classes and be composed by propagating mixin
composition. Some of these virtual classes can be defined as wrappers that serve as structural extensions of the classes of the base program. In this section we will give an overview these language features as well as explain how they support variation management in product lines.

2.2.1.1 Virtual Classes and Propagating Mixin Composition

The concept of a virtual class stems from the Beta [MMP89] programming language and was further developed in gbeta[Ern99a], which introduced propagating mixin composition [Ern99b] and a type-safe family polymorphism [Ern01]. CAESARJ integrates these concepts to Java, which has different method overriding semantics than Beta and supports abstract methods and classes.

Virtual classes are inner classes that can be refined in the subclasses of the enclosing class. They can also be considered as members of the objects of the enclosing class. We will call such objects as family objects, because they define families of objects that are instances of their virtual classes. Analogously we will refer to the enclosing classes as family classes.

A refinement of a virtual class, also known as a furtherbinding, implicitly inherits from the class it refines. In the furtherbinding we can add new methods, fields and inheritance relationships as well as override the inherited methods. In each family class all references to a virtual class are always bound to its most specific refinement.

Figure 2.1 shows an implementation of a tree data structure. The class Tree has two virtual classes Node and CompositeNode that describe correspondingly simple and composite nodes of the tree. All nodes have names, while composite nodes additionally manage references to their children.

Now imagine that we want to have a tree, the nodes of which are coloured. For this we need to define a colour attribute to its nodes and methods that work with this attribute. Since Node is a virtual class we can define ColoredTree as a subclass of Tree and define the new attribute and methods in the furtherbinding of Node as shown in Figure 2.2. The new functionality is automatically inherited by the CompositeNode of ColoredTree, because its superclass is bound to the most specific furtherbinding of Node available in ColoredTree.

Figure 2.3 shows an example of polymorphic usage of family classes, which is known as family polymorphism. The method fillTree fills a given tree with sample data. The instantiations of nodes in this method are late bound, which means that the type of the instantiated nodes depends on the dynamic type of the t parameter. The type checker of CAESARJ ensures that the nodes passed as parameters to setRoot and addChild methods are the nodes from the same family as the receiver.

This ensures that the methods, which have virtual classes as return types, return only objects from the same family too. For example, the type system ensures that in the main method of Figure 2.3 the call to findByname always returns a Node from the tree t. Since we statically know that t is a ColoredTree, we can safely call method setColor on its nodes.
public class Tree {
  protected Node root;
  public Node getRoot() { return root; }
  public void setRoot(Node n) { root = n; }
  public Node findByName(String name) { return root.findByName(name); }
}

public class Node {
  protected String name;
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
  public Node findByName(String name) { ... }
}

public class CompositeNode extends Node {
  protected List children = new ArrayList();
  public Node getChildAt(int i) { return (Node)children.get(i); }
  public int getChildCount() { return children.size(); }
  public void addChild(Node n) { children.add(n); }
  public void removeChild(Node n) { children.remove(n); }
  public Node findByName(String name) { ... }
}

public class ColoredTree extends Tree {
  public class Node {
    protected Color color = Color.black;
    public Color getColor() { return color; }
    public void setColor(Color c) { color = c; }
  }
}

public class Test {
  public static void fillTree(final Tree t) {
    t.CompositeNode n1 = t.new CompositeNode();
    n1.setName("n1");
    t.Node n2 = t.new Node(); n2.setName("n2");
    t.Node n3 = t.new Node(); n3.setName("n3");
    t.setRoot(n1); n1.addChild(n2); n1.addChild(n3);
  }
  public static void main(String args[]) {
    final ColoredTree t = new ColoredTree();
    fillTree(t);
    t.Node n = t.findByName("n1");
    n.setColor(Color.blue);
  }
}

Figure 2.1: Implementation of a tree structure

Figure 2.2: Implementation of a colored tree

Figure 2.3: An example of family polymorphism
The interfaces of family classes can be described by abstract family classes, that contain abstract definitions of virtual classes. For example, in Figure 2.4 the class `ITree` defines the read-only interface to the class `Tree`. In abstract family classes we can have concrete virtual classes with abstract methods. This means that these classes can be instantiated polymorphically over that interface.

Since family classes can contain a set of classes, they can be used instead of packages. In this way we can enjoy the benefits of inheritance, interfaces and polymorphism at the scale of sets of classes. These features are useful for implementation of large scale extensible components.

Mixin composition is a form of multiple inheritance, which is based on linearization of the inheritance graph. Inheritance linearization is a common mechanism to reduce a multiple inheritance graph to an ordered list, so that the order of the elements in the list determines the behaviour in a case of ambiguity. The C3 algorithm defines a topological sorting of a multi-inheritance graph with additional rules that make the linearization unambiguous and enforce some desirable properties of the result (see [BCH+96]). The linearization defines the overriding order of inherited methods.

CAESARJ implements a propagating mixin composition, which means that the composition propagates into virtual classes: all inherited declarations of virtual classes with the same name are composed by mixin composition. Since virtual classes may also have super classes, these are composed with mixin composition as well. The propagating mixin composition provides a large-scale multiple inheritance that allows to compose independent extensions of large-scale components. It is especially useful for feature-based decomposition as explained in the next section.

### 2.2.1.2 Modularization of Features with Virtual Classes

Since features are units of functionality they are often not aligned to the structure of software objects and operations. A state of an object may include information necessary to different features; an operation can include behaviour that serve to different features. On the other hand, multiple objects and operations can be involved in an implementation of a feature. Thus, if we want to modularize features, we should be able to split definitions of objects and operations and
distribute them into multiple modules.

Thus, modularization of features requires an appropriate module system. The modules in conventional programming languages are not extensible: they do not provide mechanisms to extend definitions of existing classes and functions. Extensibility problems are solved by various design patterns that require much additional effort and preplanning, and produce a less maintainable design.

AspectJ-like aspects [KHH+01] solve extensibility problems by supporting dynamic and static crosscutting. Dynamic crosscutting allows extending or overriding existing operations with new behaviour, while static crosscutting can be used to extend existing classes with new attributes, relationships and operations.

The problem with AspectJ aspects is that it is difficult to manage relations between them. An extension to a class introduced by an aspect is either private to the aspect or visible to all other modules. This introduces implicit dependencies between modules that are difficult to understand and to maintain. Independent development of modules very much relies on possibility to define contracts between them in form of interfaces. It is not possible to define that some aspect requires certain interface, which then must be implemented by intertype declarations of other aspects. Dynamic crosscutting also lacks contracts between the aspect and the code. An aspect can quantify over properties that are not explicitly declared, but rely on unstated assumptions such as naming conventions or organization of software into packages.

Extensible modules can be also modelled with virtual classes and mixin composition of CAESARJ. Modules are then modelled by classes (we will refer to them as family classes) and their dependencies are modelled by inheritance relations. The actual classes that implement application objects are then declared as virtual classes. Such design enables extension of classes with new attributes, relationships and operations as well as extension and overriding of existing operations. An advantage of such a module system is that the dependencies between modules are explicit, i.e. if a module class needs the extension introduced by another module class there must be inheritance relationship between them. The inheritance hierarchy also determines the order of precedence between the extensions of the modules.

Another advantage of modelling features by classes is that it provides a uniform solution at various levels of granularity: individual classes, collaborations of classes and entire application. At the level of individual classes mixin composition works as a form of multiple inheritance, which allows to modularize variations of a class. Often we need components that are larger than individual classes and are rather implemented as collaborations of multiple classes. Such components can be modelled as classes with multiple collaborating virtual classes and their variations can again be modularized by refining virtual classes, and defining various compositions of such refinements. Since the compositions are again classes, they can be instantiated at runtime. In this way we can decide at runtime, which variant of component to use. It is also possible to use multiple variants of a component simultaneously.
2.2.1.3 Dynamic Aspects

The overriding semantics of virtual classes does not provide quantification possibilities. In order to support crosscutting extensions, CAESARJ additionally provides the traditional pointcut-advice mechanism that can be used in combination with virtual classes. The pointcut language of CAESARJ is the same as that of AspectJ [KHH+01]; it provides quantification over the static structure of the program, name patterns, dynamic types of the objects related to the joinpoint (receiver, caller, arguments) and the context in the method-call stack.

One of the innovations of CAESARJ is integration of aspects and classes, which means that instead of having a special language construct for aspects, CAESARJ allows to declare pointcuts and advice directly in classes. This changes the instantiation model of aspects. While in AspectJ aspects are instantiated either as singletons or on-demand per certain pointcuts, in CAESARJ aspects are instantiated explicitly just as normal classes. As a result aspects become first-class objects: they can be referenced by variables, passed as arguments to methods and used polymorphically.

Aspect objects do not intercept joinpoints as long as they are not dynamically deployed. The difference between instantiation time and the deployment allows to initialize an aspect before it is actually used. Explicit dynamic deployment allows to deploy and undeploy the same aspect multiple times during program execution and in this way dynamically control its scope of activity. Explicit dynamic deployment is also useful for aspectual polymorphism [MO03], because the decisions when to use an aspect and what variation of aspect to use can be made independently. So dynamic aspect deployment enables dynamic variation of the crosscutting functionality modularized in aspects.

CAESARJ supports various dynamic deployment strategies that allow to limit the scope of the aspect. The simple (also known as local) aspect deployment deploys an aspect on all joinpoints on the local JVM process. Thread-local and object-local deployment strategies allow to deploy the aspects on the joinpoints of separate threads and objects. Remote aspect deployment deploys aspects on other processes and machines, which allows aspects to intercept joinpoints on other processes. Deployment strategies to bind crosscutting variation at different dynamic scopes.

2.2.1.4 Wrappers

In section 2.2.1.2 it was described how virtual classes and mixin composition can be used to modularize varying features in a product line. The problem is that such solution supports only static variation: it is not possible to activate or deactivate optional features at runtime.

Dynamic variation of crosscutting features can be achieved by dynamic aspects: pointcut and advice mechanism in combination with dynamic deployment. The problem is that pointcuts and advice can only extend the behaviour of a program, but not its static structure. Therefore, CAESARJ additionally supports the mechanism of wrappers. A wrapper is a class, which is defined as exten-
2.2.2 Steamloom

Steamloom is a virtual machine developed by TUD with explicit support for aspects. [BHMO04] In particular Steamloom provides facilities for runtime weaving and unwrapping of aspects. Runtime weaving of aspects provides various advantages over static weaving.

Runtime weaving provides direct support for dynamic aspect deployment, which otherwise should be modelled on top of static weaving. Modelling on top of static weaving is not efficient, because the aspect is woven even if it is not deployed. As a result it may cause redundant dynamic behaviour such as collection of context information about joinpoints, matching dynamic pointcuts and checking whether aspects of a particular type are deployed. Direct support of aspects in the virtual machine can also provide efficiency benefits for statically deployed aspects. [BHMO04, BKH06]

Another problem of compile-time and load-time weavers is that they do not work in a modular way: the weaver must know all possible aspects at once in order to weave them to the base program. This means that the incremental weaving is supported only with respect to new classes to be woven, but not with respect to new aspects. Weaving aspects at compile-time significantly increases compilation time, because it is time consuming process, which cannot be completely incrementalized. These problems do not exist in case of runtime weaving, because it is fully incremental: it is possible to weave both newly loaded classes as well as newly loaded aspects.

The architecture of the latest implementation of Steamloom [BM07] provides an API in form of a class library for explicit representation of pointcuts and advice. The API serves as an interface between compilers of various aspect-oriented languages and implementations of weaving on various execution environments. Currently there are implementations of the API on Sun JVM and IBM Jikes JVM.

2.2.3 Event-based AOP

2.2.3.1 The basic model

Event-based AOP (EAOP) is a general framework for AOP, introduced by Rémi Douence and Mario Südholt [DMS01], with the following characteristics:

- Aspects are defined in terms of events emitted during program execution. Events are similar to join points in AspectJ: they correspond to base actions that are about to happen.
• Crosscuts relate sequences of events, possibly including state modifications. They are defined by event patterns, which are matched during program execution.

• Once a crosscut has been matched, an associated action is executed.

The main assumption behind EAOP is that crosscut definitions should be expressive enough to relate different events occurring during program execution and make explicit the state information belonging to those events. This in contrast to other approaches to AOP (most notably AspectJ), which restrict crosscuts (pointcuts in AspectJ) to individual points during program execution and the state associated to them is the information local to those execution points. This model is general enough to accommodate, in principle, any other model of AOP (with respect to behavioural crosscutting).

An interesting instance of EAOP defines aspects as finite labelled transition systems where labels have the following form: event > action. The details of how events and actions are built may vary depending on the details of the considered event and action languages. Let us consider a simple example, where the events are simply identifiers and actions sequences of primitive actions (also identifiers), with the additional constraint that each action sequence contains either the primitive action skip or the primitive action proceed. The action skip means that the base action corresponding to the initial event must be skipped and the action proceed means that this base action must take place. In this setting, Figure 2.5 shows an EAOP aspect, called Consistency, that is interested in update events taking place between a login and a checkout event. The matching update actions are skipped and replaced by a log action. The aspect does not care of updates taking place after a checkout and before a login. These updates are therefore left unaffected by the aspect.

This leads to an intuitive interpretation and representation of aspects and facilitates formal reasoning, for instance the analysis of aspect interactions [DFS02a, DFS04b].

2.2.3.2 Concurrent AOP

The starting point of Concurrent AOP (CEAOP) [DLBNS06] is the possibility of representing processes as labelled transition systems, which makes it possible to use a common framework to model processes and aspects, and leads to an elegant model of concurrent aspects.
Concurrent aspects are described, as EAOP aspects, by labelled transition systems, whereas base programs are described by simpler (finite) labelled transition systems, whose labels are simply (primitive) actions represented by identifiers. The labelled transition systems used for representing base programs correspond to the labelled transition systems (LTSs) of Magee and Kramer [MK06], with their textual representation as Finite State Processes (FSP). Figure 2.6 shows the FSP model of a simple e-commerce server. A customer can log in, browse the catalogue of the store and check out. A store administrator can update the catalogue at anytime. The aspect described in Figure 2.5 can then be composed with such an application.

The semantics of the composition is given by two different transformations combined with standard LTS parallel composition. The first transformation, Instrument, turns the base LTS into an instrumented LTS. The second transformation InsertWLoops, turns the aspect LTS into a plain LTS by inserting waiting loops. A parallel composition of the two LTSs models the complete system. This complete system can then be verified against safety and progress properties using the Labelled Transition System Analysers (LTSA) [MK06, MKCU].

The transformations are fully described in [DLBNS06]. Let us just give the intuition behind these transformations using the above example.

The result of the transformation by Instrument of the base LTS is given in Figure 2.7. The role of the transformation is twofold: generate events for actions of interest and make these actions controllable by aspects. Concretely, a bUpdate event indicating that an update is about to happen is inserted before each update action. As an aspect interested in an update may decide to skip or not to skip the update, a choice is immediately added after the bUpdate event.

The result of the transformation by InsertWLoops of the aspect LTS is given in Figure 2.8. The initial definition of the aspect only talks about the events of interest in each state. A complete LTS model has to define what happens for each event in each state. This is similar to the completion of an LTS representing

\[ \text{Server} = (\text{login} \rightarrow \text{Session} | \text{update} \rightarrow \text{Server}), \text{Session} = (\text{checkout} \rightarrow \text{Server} | \text{update} \rightarrow \text{Session} | \text{browse} \rightarrow \text{Session}). \]

**Figure 2.6:** An FSP model of a simple e-commerce server

---

1 For those who are not familiar with LTS composition, let us remind that if LTSs \( l_1 \) and \( l_2 \) have states in \( S_1 \) and \( S_2 \), the states of \( l_1 \parallel l_2 \) are elements of \( S_1 \times S_2 \). The synchronization of the two LTSs is based on shared actions. A transition from a state \((s_1, s_2)\) with action \( a \) is only possible in two cases:

- There is a transition with action \( a \) either from \( s_1 \) in \( l_1 \), or from \( s_2 \) in \( l_2 \) and \( a \) does not belong to the alphabet of the other LTS.
- There is a transition with action \( a \) both from \( s_1 \) and \( s_2 \).
Server =
  ( login -> Session
   | bUpdate -> ( skip -> Server
       | proceed -> update -> Server
   )
  ),
Session =
  ( checkout -> Server
   | bUpdate -> ( skip -> Session
       | proceed -> update -> Session
   )
   | browse -> Session
  ).

**Figure 2.7:** Transformed base FSP

Consistency =
  ( login -> Session
   | bUpdate -> proceed -> Consistency
   | checkout -> Consistency
  ),
Session =
  ( bUpdate -> skip -> log -> Session
   | checkout -> Consistency
   | login -> Session
  ).

**Figure 2.8:** Transformed aspect FSP
Figure 2.9: Model of the woven system

\[
\begin{align*}
    b\text{Update} & \rightarrow \\
    & \quad ( b\text{Skip} \rightarrow e\text{Skip} \rightarrow e\text{Update} \rightarrow \text{Server} \\
    & \quad \mid b\text{Proceed} \rightarrow \text{update} \rightarrow e\text{Proceed} \rightarrow e\text{Update} \rightarrow \text{Server} )
\end{align*}
\]

Figure 2.10: Enhanced instrumentation

A security property: new transitions are created in each state for each (shared) label not labelling an outgoing transition. These transitions lead to a predefined error state. In the case of an aspect, two different cases have to be dealt with:

- The missing label corresponds to a **skippable** action of the base program. Additional transitions should not be built for this action, but for the corresponding event, that is \texttt{bUpdate}, rather than \texttt{update} in our example. Also, if there was no transition specified for this event in a given state, this means that the corresponding base action should be left undisturbed: a \texttt{proceed} action is generated before looping back to the initial state (see the additional branch created for \texttt{bUpdate} in the Consistency state).

- The missing label corresponds to an action that is **not** skippable. This is the case of \texttt{login} and \texttt{checkout} in our example. The added transitions are simpler, a new branch is created for each missing label, looping to the initial state.

The actions of the aspect are also turned into plain LTS labels.

The result of the composition of the base and aspect LTS is given in Figure 2.9. One can see in particular that \texttt{browse} and \texttt{log} actions are interleaved. This corresponds to a concurrent execution of the base program and the system.

**Controlling concurrency between base and aspect**  
Controlling concurrency between the base program and the aspect requires adding more synchronization events. Figure 2.10 shows the full instrumentation scheme for the \texttt{update} action (it is also used for aspect composition).
Switching between a situation where the aspect and the base run sequentially (Figure 2.11) or in parallel (Figure 2.12) is just a matter of hiding the aspect eUpdate action, marking the end of the aspect, or not. When the action is hidden (an operation that takes place prior to performing parallel composition), it is not considered any longer as a shared action: the eUpdate action of the base instrumentation can take place without waiting for the aspect to terminate. Note that the synchronization arrows (between the base program and the aspect) are directed in order to show the actual meaning of the example.

Composing aspects  Aspects are basically composed using parallel composition. But plain parallel composition is not enough and is just a very specific form of composition, which disallows, for instance, that an aspect proceeds while another skips a base action (this would result in a deadlock). Additional composition operators can be provided as processes plugged, through parallel composition and proper renaming, onto the aspects and the base program.

Figure 2.13 shows a simplified structural representation of the Fun operator (some instrumentation actions are ignored), which implements a sequential composition of two aspects Aspect1 and Aspect2. The processes, i.e., the base program, the aspects and the operator process, are here represented as some kind of components with a hidden implementation (except that we show the implementation of the process associated to the operator) and an interface con-
sisting of their alphabet. The composition consists then in properly connecting the interfaces, which may require to rename some of the labels. For instance, the process operator includes two actions $skip_1$ and $skip_2$ corresponding to be connected to the $skip$ actions of the composed aspects, which have therefore to be renamed. Again, the arrows are here to give the intuition behind the connection (the aspects, not the process operator, initiate skipping).

What happens with such an operator is that either both aspects proceed and the base action is executed (the corresponding control flow is given in Figure 2.14) or the base action is skipped. Moreover, each aspect takes its decision sequentially, first $Aspect_1$ and then, if $Aspect_1$ decides to proceed, $Aspect_2$.

Figure 2.15 shows another composition with an operator $ParAnd$, whereby the decision is taken in parallel and the base action is executed if and only if $Aspect_1$ and $Aspect_2$ decide so.

From models to concrete implementations Baton [NN07b] is a small language that implements CEAOP. The syntax of Baton is very close to FSP, and can be compiled into Java. Baton aspects and components are seamlessly integrated in a very basic component model, in which a component implements a simple interface and defines a protocol. A Baton aspect, as well as a component, implements an interface and defines a protocol. This protocol has however a slightly different meaning than a standard component protocol. It corresponds to the definition of a stateful concurrent aspect, which can observe various base actions (service requests and replies, internal actions) and react accordingly. These base actions are defined in the interface as either (non-
Figure 2.14: The control flow resulting of the application of the Fun operator

ParAnd = (skip1 -> (skip2 -> skip -> ParAnd
| proceed2 -> skip -> ParAnd
| proceed1 -> (skip2 -> skip -> ParAnd
| proceed2 -> proceed -> ParAnd)).

Figure 2.15: Structural representation of the application of the ParAnd operator
aspect Consistency implements IConsistency {
    OutSession = ( login -> Session ),
    Session = ( update > skip; log -> Session | checkout -> OutSession ),
    event login();
    event checkout();
    skippable event update();
    action log();
}

Figure 2.16: Example of an aspect in Baton

skippable) event or skippable event. The actions used to react to events are defined as action. Figure 2.16 shows an implementation of our previous Consistency aspect (see Figure 2.5) in Baton. Finally, Baton aspects can be instantiated and their abstract actions connected to concrete base actions. An earlier paper shows [NN07a] how events can be parameterized and gives some details on the machinery required to implement synchronization on shared events.

2.2.3.3 Distributed AOP

Distributed applications are inherently more complex to develop than sequential ones because of the additional requirements brought about by a partitioning of the software system across a network (e.g., handling of data replication, communication and synchronization between system components, ...) and the crosscutting nature of distribution concerns with respect to the other functionalities of product lines or other software systems. Techniques developed in the field of Aspect-Oriented Software Development (AOSD) [KLM+97] should be useful to separate these distribution concerns. However, despite its increasing popularity for sequential applications, relatively few AO approaches address the development of distributed software.

In this section, we introduce the AWED system ("Aspects with Explicit Distribution") [BNSV+06] that provides the most comprehensive support for the modularization of distribution concerns using aspects. We concentrate, in particular, on three issues that are of particular interest in the context of the AMPLE project:

- The language features for distribution (and concurrency) that AWED provides.
- A comparison between the distribution of CaesarJ and AWED.
- A notion of invasive patterns for distributed programming that can be directly supported using the modularization mechanisms AWED provides.
The AWED language  Modularization of crosscutting concerns for distributed applications using a domain specific aspect language, i.e., in terms of pointcut, advice and aspect abstractions, suggests support for the following issues: (i) a notion of remote pointcuts allowing to capture relationships between execution events occurring on different hosts, (ii) a notion of groups of hosts which can be referred to in pointcuts and manipulated in advice, (iii) execution of advice on different hosts in an asynchronous or synchronous way and (iv) flexible deployment, instantiation, and state sharing models for distributed aspects.

AWED provides such support through three key concepts at the language level. First, remote pointcuts [NST04], which enable matching of join points on remote hosts and include remote calls and remote cflow constructs (i.e., matching of nested calls over different machines). As an extension of previous approaches AWED supports remote regular sequences which include features of other recent approaches for (non-distributed) regular sequence pointcuts [DFS02b, DFS04a, DFL+05, A+05, VSCDF05]. Second, support for distributed advice: advice can be executed in an asynchronous or synchronous fashion on remote hosts and pointcuts can predicate on where advice is executed. Third, distributed aspects, which enable aspects to be configured using different deployment and instantiation options. Furthermore, aspect state can be shared flexibly among aspect instances on the one hand, as well as among sequence instances which are part of an aspect on the other hand.

Pointcuts  Pointcuts are basically built from call constructors (execution allows to denote the execution of the method body), field getters and setters, nested calls (cflow) and sequences of calls.

AWED employs a model where, upon occurrence of a join point, pointcuts are evaluated on all hosts where the corresponding aspects are deployed. Pointcuts may then contain conditions about (groups of) hosts where join points originate (term host(Group)), i.e., where calls or field accesses occur. Furthermore, pointcuts may be defined in terms of where advice is executed (term on(Group)). Groups are sets of hosts which may be constructed using the host specifications localhost, jphost and adr:port, which respectively denote the host where a pointcut matches, the host where the corresponding join point occurred and any specific host. Alternatively, groups may be referred to by name. (Named groups are managed dynamically within advice by adding and removing the host which an aspect is located on.)

As a first example, the following simple pointcut could be part of a replicated cache aspect:

```java
call(void initCache()) && host("ClusterGroup")
```

Here, the pointcut matches calls to the cache’s initCache method that originate from the host that has the specified address. The advice will be executed on any host where the aspect is deployed (possibly multiple ones) as there is no restriction on the advice execution host. The following example restricts the execution hosts to be different from the host where the joinpoint occurred:
pointcut putCache(Object key, Object o):
  call(* Cache.put(Object,Object))
  && lon(jphost) && args(key, o)

Here, the pointcut matches calls to the cache’s put operation on hosts other than the joinpoint host and binds the corresponding data items. (Note that in this case the clause !host(localhost) could replace !on(jphost) to achieve exactly the same effect of matching non-local joinpoints.) If the corresponding advice puts the item in the local cache, a condition on the aspect type (named, e.g., ReplCache), such as !within(ReplCache), can be used to avoid triggering the pointcut during the advice execution.

Sequences  Sequences are supported by two constructions on the pointcut level. First, the pointcut seq allows to define sequences which may be named using an identifier and consist of a list of (potentially named) transitions. A sequence matches if one of the current accepted transitions matches the current joinpoint. Advice can be triggered after a specific transition within a sequence using a term of the form s: seq(... l: logout() ...)&& step(s, l) where s and l are labels for the sequence and the transition respectively.

To illustrate the use of sequence pointcuts, consider the following pointcut, which could be part of a simple cache replication protocol:

pointcut replPolicy(Cache c):
  replS: seq(s1: initCache() && target(c) → s3 || s2 || s4,
  s2: cachePut() → s3 || s2 || s4,
  s3: stopCache() → s1
  s4: cachelInconsistency())

(Here, identifiers like initCache denote undefined pointcuts specifying corresponding call pointcuts.) The pointcut above defines a sequence of four steps. An initialization step which may be followed either by a put operation (s2), termination of the cache (s3) or an error step (s4). A put operation (s2) may be repeated, followed by a cache termination (s3) or result in a cache inconsistency (s4). After cache termination, the cache may be initialized once again. Finally, a cache inconsistency terminates the sequence (and may be reacted upon by advising the pointcut).

Advice  Advice is mostly defined as in AspectJ: it specifies the position where it is applied relative to the matched join point, a pointcut which triggers the advice, a body constructed from Java statements, and the special statement proceed (which enables the original call to be executed).

In an environment where advice may be executed on other hosts (which is possible in AWED using the on pointcut specifier), the question of synchroniza-

\[^2\]Note that while our sequences obviously encode finite-state automata, many applications of regular structures, in particular communication protocols [DFL+05], are effectively sequence-like, i.e., of a one-dimensional directed structure, so that we decided to use the more intuitive terminology for AWED.
tion of advice execution with respect to the base application and other aspects arises. AWED supports two different synchronization modes for remote advice execution: by default remote advice is executed asynchronously to the calling context. In this case synchronization, if necessary, has to be managed by hand. Local advice is always executed synchronously. The semantics of the proceed statement is “localized”: the last around advice invokes the original behaviour on the local host. The return value of the around advice is sent back to the original joinpoint host and processed in the regular around advice chain on that joinpoint host in case of a synchronous advice execution. Asynchronous advices are executed in parallel and there is thus no guarantee with respect to advice precedence. We have opted for this semantics because it provides for an intuitive yet efficient remote advice execution semantics.

Advice is also used to manage named groups of hosts: addGroup adds the current host to a given group, removeGroup allows to remove the current host from a group.

To give an example of basic advice functionality, the following advice definition is useful in the context of collaborating replicated caches:

```java
around(String k, Object o): putCache(k, o) {
    Object obj = getNewRemoteValue(k);
    if (obj != null) { proceed(k, obj); }
    else { proceed(k, o); }
}
```

This advice first tests whether a new value is present remotely for a given key. If this is the case, the new value is stored in the cache.

**Aspects**

Aspects group a set of fields as well as pointcut and advice declarations. Aspects may be dynamically deployed on all hosts or only the local one.

Furthermore, aspects support four instantiation modes: similar to several other aspect languages, aspects may be instantiated per thread, per object, or per class. However, aspect instances may also be created for different sets of variable bindings arising from sequences (term perbinding) as introduced in [DFL+05, A+05]. In this last case, a new instance is created for each distinct set of bindings of the variables in the sequence, i.e., of the variables declared as arguments of a sequence pointcut or fields used in the sequence pointcut.

Finally, AWED allows distributed aspects of the same type to share local state: values of aspect fields may be shared among all aspects of the same type, all instances of an aspect which have been created using the instantiation mechanisms introduced before, all aspects belonging to the same group or all aspects on the one host (note that these possibly belong to different execution environments, such as JVMs).

**Comparison of CaesarJ and AWED**

Support for distributed aspects is also available in CAESARJ in the form of remote aspect deployment. The approaches of AWED and CaesarJ are very different: AWED provides a set of a dedicated
language mechanisms to describe distribution, while the remote deployment in CaesarJ is just one of the deployment strategies, which is provided over the runtime library rather than by a new language construct. The advantage of having dedicated language constructs is that certain functionality can be expressed more concisely. On the other hand each new construct makes the language more complicated. This trade-off can be resolved by providing the language features in form of DSLs (see Sec. 2.3.8).

Aspects in CAESARJ are unaware of their deployment strategies, which means that they do not make any assumptions about their usage in a distributed setting, while in AWED the specification of distribution is part of the aspect definition. The both approaches have their advantages and disadvantages. In CAESARJ the distribution is more transparent, because the same aspects can be used locally as well as remotely. For instance, it is possible to transition from a standalone solution to a distributed solution without changing existing aspects. However, remote pointcuts are necessary when want to quantify over the distributed properties of joinpoints or to relate the joinpoints from different hosts. Such quantification is not possible in CAESARJ.

CAESARJ also uses different aspect instantiation and state sharing model. AWED is based on implicit instantiation of aspects similarly as in AspectJ: the aspect is instantiated automatically on certain scopes and the instantiation model is specified directly in the aspect. In CAESARJ the aspects are instantiated explicitly in the same way like classes, which also implies that the instantiation is specified outside aspects. While the implicit instantiation is more declarative, the explicit instantiation provides more flexibility.

The differences of state sharing are analogous. While in AWED state sharing model is declared in the aspect, in CAESARJ state sharing is determined outside aspect by explicit aspect instantiation and deployment. For example, a developer may chose whether the same aspect instance is deployed on multiple hosts, or for each host a separate instance of an aspect is created and deployed.

Invasive patterns The complexity of the development of today's large software systems is often manifest in a wide gap between well-structured conceptual architectures and difficult-to-manage implementations. This problem is much aggravated in the presence of crosscutting concerns.

Architectural descriptions, especially of the runtime behaviour of software systems, are frequently defined in terms of communication and communication patterns. In concurrent, parallel and distributed environments, patterns such as farming-out computations to a set of nodes, gathering data from several nodes on another nodes and pipelining are frequently used for this purpose.

In the presence of crosscutting concerns, such simple computation and communication patterns frequently are not sufficient anymore because the application of such a pattern then depends on crosscutting accesses to execution state. In performance critical applications, such as caching infrastructures in distributed systems, pattern applications are then copied and slightly modified
In order to address this problem, the notion of “invasive patterns” [BNSDM07] has been developed. These patterns generalize standard distributed and/or concurrent computation and communication patterns by capabilities for the modularization of crosscutting pattern-enabling conditions using stateful aspects.

The resulting notion of invasive patterns is illustrated in Fig. 2.17 for the case of a gather pattern. On the three nodes on the left hand side, different pointcuts (represented by dashed lines) are used to access information that is then prepared by “source” advice (represented by the filled rectangles) to be sent to the right hand side node. Once all relevant data has been passed to the right hand side node, a “target” advice is used to integrate the transmitted data with an existing or new computation on the target node. In order to support the declarative definition of such crosscutting accesses, stateful pointcut languages are used that enable matching of sequences of execution events to be defined using expressive languages, in particular finite-state automata.

Support for definition of invasive patterns on the language level has been defined in form of a pattern-defining DSL [BNSDM07] that uses a simple group definition language to define source and target nodes of an invasive patterns and that incorporates aspect definitions to be applied to source and target nodes.

An instantiation of invasive patterns for distributed programming has been realized in terms of the AWED system for distributed aspects (cf. Sec. 2.2.3.3): in this implementation stateful distributed aspects are used to modularize state accesses on sets of source hosts, send information to a set of target hosts and use remote advice to integrate the data sent on the target hosts.

The notion of invasive patterns for distributed programming has proven worthwhile in the context of the modularization of two non-trivial crosscutting concerns of JBoss Cache, an infrastructure for replicated transactional caching in large-scale Java-based distributed applications. Concretely, these patterns have enabled the direct implementation of a conceptual runtime architecture defined in terms of the composition of four pattern applications [BNSDM07].
2.2.4 openArchitectureWare

openArchitectureWare (oAW) is an open source MDSD framework implemented in Java and integrates a number of tool components. oAW supports arbitrary import model formats, meta models, and output code formats. oAW is integrated into Eclipse and provides various plugins that support model-driven development. It contributes to and reuses components from the Eclipse Modeling Project [Ecl07a].

At the core, there is a workflow engine allowing the definition of transformation workflows by sequencing various kinds of workflow components. oAW comes with pre-built workflow components for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code. oAW provides a family of specialized languages for specifying constraint checks, transformations and code generators. All of those languages are built on a common OCL-like expression language. Editors and debuggers integrated into Eclipse are provided for all those languages.

Figure 2.18 gives an overview of oAW and its main building blocks. The following list explains the different components (marked with numbers):

1. Model verification using constraints: Both models and meta models can be checked for validity. In oAW constraints are defined using the Checks language.

2. Artifact generation: Any textual artifact can be generated from models using the Xpand template language.

3. Integrating generated code with manually written code: The Recipe Framework can be used to enforce rules on the code base. For example, a generator can generate an abstract base class from which developers extend their implementation classes containing the business logic. Such rules can be specified using Recipes.
4. Model modification: Models can be modified or completed using the Xtend language.

5. Model transformation: Models can be transformed into other models using the Xtend language. Typically, input and output models are instances of different meta models.

6. Loading and storing models: By default, EMF models are stored in XMI files. oAW provides workflow components for loading and storing models from/to XMI files.

7. Model creation using UML tools: Models can be created and edited using familiar UML tools such as Rational Rose or MagicDraw. oAW provides adapters for various commonly used UML tools.

8. Textual model creation: oAW can also read textual models created with custom-built textual editors based on its Xtext framework. The DSL is described in EBNF and the parser, meta model, and customized editor is created automatically.

9. oAW also integrates with custom graphical editors built using Eclipse GMF. oAW constraints can be evaluated inside GMF editors in real time, and oAW can process models created with GMF.

2.2.4.1 Integrating oAW and pure::variants

pure::variants is a variant management tool that manages product line variability and assists in managing and assembling individual product variants. The basic idea of pure::variants is to realize product lines by feature models and family models. The problem space is captured with feature models and the solution space is captured with family models, separately and independently. A family model consists of so-called components. Components represent at least one attribute of the software solution and contain the logical parts like classes, objects, functions, variables, and documentation. A feature model specifies the interdependencies of the features of a product line. They represent all variability of the products in the product line. pure::variants also supports the use of multiple feature models that are hierarchically linked.

Users can select features required for the desired product from the feature models. A configuration model represents such a selection of features. pure::variants checks the validity of this selection and if necessary automatically resolves dependency conflicts. A valid selection triggers an evaluation of the family models that contain component definitions consisting of logical and physical parts. The evaluation process results in an abstract description of the selected solution in terms of components.

Within the oAW tooling, global configuration models can be queried and hence, MDSD activities can be performed based on the selection of features. Workflow components, model-transformations, and code generation templates can
be linked to features. Their execution then depends on the presence or absence of features in the configuration model.

As illustrated in Figure 2.19 (a), the data transfer between pure::variants and oAW is done using EMF Ecore. An automatic variant model export has been integrated into pure::variants. The oAW runtime reads this model and makes the variant information available for querying by oAW workflows, transformations and templates.

The integration of pure::variants and oAW also supports addressing properties or attributes of features. The values of properties and attributes can be read and used in the transformation or code generation templates.

### 2.3 Extensions and Improvements

#### 2.3.1 Contracts between Features

**Motivation** When modularizing features by family classes as described in 2.2.1.2 dependencies between features are modelled by inheritance relationships of the corresponding family classes. Such solution is not very modular, because the client feature depends on the implementation of its supplier. In order to increase modularity we can define an explicit interface of the supplier feature by defining an abstract family class that declares abstract definitions of classes and their methods that should be visible to the clients. In CAESARJ this is enabled by support for abstract virtual classes. It is even possible to have segregated interfaces that are targeted to different types of clients. Then the implementation of the client feature would inherit from the abstract class instead of the implementation.

The separation of interface and implementation has a further advantage: we can provide alternative implementations of an interface and in this way make the clients of the interface composable with each implementation. This can be very useful for implementation of alternative features.

The problem is that simple inheritance relations are not able to precisely describe contracts between features. The client of an interface inherits from it in the same way as its provider: through a simple inheritance relation. Their contracts are however different: the client should use the methods declared in the interface, while the provider should implement these methods. We can force the
provider to implement the methods of the interface by declaring it as a concrete class, while the clients of the interface can be declared as abstract classes. The problem is that the provider of the interface can at the same time be a client of another interface, thus, it should also be declared as abstract. So we need a way to declare classes that are concrete with respect to the interfaces that they provide, but still abstract in general, because they are clients of some other interfaces.

Proposal  In order to address the problem of different contracts of clients and suppliers of an interface, we propose to introduce requires relationship and to change the rules of checking classes for abstractness.

If a class is concrete (not marked as abstract) it must implement all abstract methods except for those that are inherited over the requires relationships. Analogously it must implement the abstract methods of concrete virtual classes that are not inherited over the requires relationships.

A feature class that provides an interface simply extends from it, while a client of the interfaces includes it in its requires clause. Now if we declare both the provider and the client as concrete classes, the type checker will check if the provider completely implements all the methods of the interface, while the client must implement all methods except those from the interface.

Inheritance relationships of superclasses are propagated in a different way:

- If A extends B and B extends C, then A extends C,
- If A extends B and B requires C, then A requires C,
- If A requires B and B requires C, then A requires C,
- If A requires B and B extends C, then A requires C.

Extends relationships eliminate indirect requires relationships. If A requires B and A extends B, directly or indirectly, then the B requirement is eliminated and A only extends B.

Since we have changed the meaning of the abstract keyword, we must also change the rule when a class can be instantiated. The classes that are not marked as abstract can still inherit abstract methods through their requires relationships. Thus, it would be not type safe to allow their instantiation. Only the concrete classes without requires relationships can be instantiated. We will call such classes as complete and the classes with unfulfilled requires relationships as incomplete. It can be automatically inferred whether a class is complete and thus whether it can be instantiated, but we could also provide a possibility to mark a class by a complete keyword and in this way force the compiler to check if the class is actually complete.

Note that we don’t have explicit constructs for interfaces so in the requires clauses we would list abstract classes. The need for explicit construct for interfaces is questionable, because they can be declared as completely abstract.
classes. Besides, virtual interfaces would create a problem for support of dynamic composition of classes as it is explained in the next section.

Another problem with interfaces is that they cannot be extended with concrete methods, because would be a problem for support of dynamic composition of classes.

2.3.2 Private Inheritance

Motivation Another problem with modelling feature dependencies by inheritance relationships is related to the transitivity property of the inheritance relation: a class can use the functionality of not only its direct parents, but also the functionality of the parents of parents. It is different from typical module systems: the dependencies of a module class are normally considered as part of its implementation and are not exposed to its clients. Transitive dependencies reduce understandability, because it is difficult to see what of the transitive dependencies of a module class are actually used, and stability, because by changing dependencies of a module class we can potentially destabilize its clients.

Proposed Solution To address the problem of dependency on transitive inheritance relationships, we propose to introduce an inheritance relationship, which is private to a class in the sense that it is not visible in the subclasses of the class. We will denote this relationship as usage and declare it in a special uses clause.

The usage relationship is not propagated to subclasses: if class A uses class B, then A can access the definitions of B, but not its subclasses. The extends relationships of B can be used to eliminate requirements of A, but not the requirements of the subclasses of A. The requirements of B do not automatically propagate to subclasses of A either, which means that they must be either eliminated by other relationships of A or they must be explicitly redeclared as requirements of A.

Usage relationships of a class can be considered as its implementation details, because adding or removing them could not influence type safety of the clients of the class.

2.3.3 Extensible Pointcut Abstractions

Motivation Aspects support extension of existing classes and their methods. However, in certain cases it may be necessary to extend definitions of aspects. For example, imagine that we want to implement an aspect which tracks changes in the data model of an application in order to know if data model has to be saved. Let’s assume that our data model consists of two classes Event (see Figure 2.20) and Person (see Figure 2.21). A straightforward solution is to write an aspect with a pointcut that gathers all methods of Event and Person that change the state of the objects of the classes (see Figure 2.22).
package events;

public class Event {
    private String title;
    private Date date;
    public Date getDate() { return date; }
    public void setDate(Date date) { this.date = date; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}

Figure 2.20: Event class

package persons;
import events.Event;

public class Person {
    private int age;
    private String name;
    private List<Event> events = new LinkedList<Event>();
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public Iterator<Event> getEvents() { return events.iterator(); }
    public void addEvent(Event e) { events.add(e); }
    public void removeEvent(Event e) { events.remove(e); }
}

Figure 2.21: Person class

Now imagine that Event and Person are just two classes of a large data model that consists of a hundreds of classes. In this case the pointcut of our aspect would specify all methods that change the data model. Such solution is disadvantageous from the perspective of extensibility, because every time we decide to extend or change the classes of our data model we have to change this aspect. Such aspect would also make a problem for variability management in a context of a product line. Since different classes and methods of the data model are related to different features, modularization of the aspect by features would require splitting the pointcut definition so that each part of the pointcut selects only the methods of one feature.

One way to solve this problem in AspectJ would be to define an abstract

package persistence;
import events.Event;
import persons.Person;

public aspect ChangeTracker {
    private boolean changed = false;
    void reset() { changed = false; }
    pointcut modelChange() :
        execution(void Event.set(..)) ||
        execution(void Person.set(..)) ||
        execution(void Person.addEvent(Event)) ||
        execution(void Person.removeEvent(Event));
    after(): modelChange() { changed = true; }
}

Figure 2.22: Simple aspect to track changes in data model
aspect with an abstract pointcut and a set of concrete aspects that specify the pointcut for each feature. Figure 2.23 demonstrates a solution for our example. ChangeTracker is now an abstract aspect, which defines the functionality of change tracking and abstracts from the methods to be tracked by defining an abstract pointcut. Then we define concrete subaspects of ChangeTracker that specify the pointcut for each feature.

Such solution is also not without problems. Note that we transformed our aspect into multiple aspects and made the fields and operations of ChangeTracker static, because they must be shared between multiple concrete subaspects. This makes much more difficult to control the tracking functionality dynamically in the context of CAESARJ or another programming language that supports dynamic aspect deployment, because now we have to manipulate multiple aspects rather than one. The static state also does not support multiple instances of the aspect.

Another problem is that the pointcut modelChange that specifies what methods cause changes in the data model is tightly coupled to concrete change tracking functionality. It is not possible to vary the change tracking functionality or to reuse the pointcut for different purposes, for example for update of display or enforcing read-only mode in the application.

With introduction of annotations in AspectJ 5, new solutions to the problem are possible. Instead of listing the methods that change the model in a pointcut, we can mark each of these methods with a specific annotation and then to write a pointcut, which collects all methods with that annotation. Such solution is demonstrated in Figure 2.24.

The problem with the latter solution is that the change tracking concern is not completely separated from the model classes. However, this problem can...
package events;
import java.util.Date;
import persistence.ChangesModel;

public class Event {
    private String title;
    private Date date;
    public Date getDate() {
        return date;
    }
    @ChangesModel
    public void setDate(Date date) {
        this.date = date;
    }
    @ChangesModel
    public String getTitle() {
        return title;
    }
    @ChangesModel
    public void setTitle(String title) {
        this.title = title;
    }
}

package persistence;

public aspect ChangeTracker {
    boolean changed = false;
    void reset() {
        changed = false;
    }
    protected pointcut modelChange() :
        @annotation(ChangesModel) &&
        execution(* (*(..)));
    after() : modelChange() {
        changed = true;
    }
}

Figure 2.24: A solution with annotations

package events;
import persistence.ChangesModel;

public aspect EventChanges {
    declare @method: void Event.set(*(..)) : @ChangesModel;
}

package persons;
import persistence.ChangesModel;
import events.Event;

public aspect PersonChanges {
    declare @method: void Person.set(*(..)) : @ChangesModel;
    declare @method: void Person.addEvent(Event) : @ChangesModel;
    declare @method: void Person.removeEvent(Event) : @ChangesModel;
}

Figure 2.25: Adding annotations with aspects

be solved using intertype declarations of annotations: instead of writing annotations directly inside data model classes, we can write a set of aspects that introduce these annotations as shown in Figure 2.25.

Aspects with annotations solve the stated modularization problem. However, the solution employs a set of complicated language features. In spite of that its expressivity is still limited. Annotations can be used to mark a set of classes, methods or fields, but in general a pointcut can be more complicated than a simple selection of a set of methods. It can include additional conditions, for example constrain the types of method parameters or add some conditions on the control flow. Annotations also do not provide the same composability as pointcuts: an annotation cannot be defined as a combination of other annotations. On the other hand, annotations can express sets that cannot be expressed by pointcuts, because pointcuts can express only sets of joinpoints. AspectJ use various patterns that select methods or types inside pointcut definitions, but
these patterns are not available as first class values.

**Proposed Solution**  The proposed solution is based on two ideas: to make various patterns that are used in pointcut definitions as explicit entities in the program and to apply mixin composition semantics to pointcuts and their patterns in order to make them extensible and composable in a similar way like virtual classes.

Figure 2.26 demonstrates a possible design of our example with extensible pointcuts. Here we modularize features using virtual classes and propagating mixin composition: each top level class represents a feature module, the dependencies between features are expressed by inheritance relationships. Module MModelChanges declares pointcut modelChange, which should define all the points where the data model is changed. The pointcut definition in this module is empty and is extended in the modules MEventChanges and MPersonChanges for the corresponding features. The pointcuts modelChange of these modules are defined as unions of the super pointcut and the pointcuts that specify the model changes in the corresponding feature. In a concrete application, which is defined as a composition of feature modules, the pointcuts are composed using mixin composition semantics.

Figure 2.27 shows an alternative solution in a language supporting various extensible sets. For example, we can assume that in a lot of cases, the methods that change data model match one of the name patterns set*, add* and remove*. Then we can declare defaultChange as a set of classes for which it is the case. Then the modules that define model changes in particular features can specify can extend the set of classes with default change methods, as well as extend the pointcut modelChange for other cases.

### 2.3.4 Extension Interface

**Motivation**  Aspect-oriented programming languages provide a lot of flexibility for extending programs. For example, AspectJ allows changing behaviour of software at any method call and field access without any constraints. There are, however, various disadvantages of such flexibility. A provider of a reusable module cannot be sure that the clients use it only over the explicitly defined interface, because a client can define an aspect that bypasses the interface and advises internal methods and fields of the module. As a consequence, the provider of the module can not know if a change to the implementation to the module won’t break the code of its clients even if the interface of the module does not change. Besides, the clients also cannot know what kind of extensions with aspects wouldn’t break the correctness of the module. The lack of explicit interfaces for aspects also complicates modular reasoning, because in order to determine the properties of a module or to understand the impact of its change we must know all aspects in the system that can potentially advise this module. [KM05]
abstract public cclass MEvents {
    public cclass Event {
        private String title;
        private Date date;
        public Date getDate() { return date; }
        public void setDate(Date date) { this.date = date; }
        public String getTitle() { return title; }
        public void setTitle(String title) { this.title = title; }
    }
}

abstract public cclass MPersons extends MEvents {
    public cclass Person {
        private int age;
        private String name;
        private List<Event> events = new LinkedList<Event>);
    ... 
}

abstract public cclass MMModelChanges {
    pointcut modelChange() : /\ empty */;
}

abstract public cclass MChangeTracker extends MMModelChanges {
    boolean changed = false;
    void reset() { changed = false; }
    after() : modelChange() { changed = true; }
}

abstract public cclass MEventChanges extends MEvents & MMModelChanges {
    pointcut modelChange() : super.defaultChange() || execution(void Event.set(..));
}

abstract public cclass MPersonChanges extends MPersons & MMModelChanges {
    pointcut modelChange() :
        super.defaultChange() ||
        execution(void Person.set(..)) ||
        execution(void Person.addEvent(Event)) ||
        execution(void Person.removeEvent(Event));
}

class MApplication extends MPersonChanges & MEventChanges & MChangeTracker & ... { }

Figure 2.26: A solution with extensible pointcuts

abstract public cclass MMModelChanges {
    pointcut modelChange() :
         execution(void defaultChange().set(..)) ||
         execution(void defaultChange().add(..)) ||
         execution(void defaultChange().remove(..));
    set class defaultChange() :
}

abstract public cclass MEventChanges extends MEvents & MMModelChanges {
    set class defaultChange() : super.defaultChange() || Event;
}

abstract public cclass MPersonChanges extends MPersons & MMModelChanges {
    set class defaultChange() : super.defaultChange() || Person;
}

Figure 2.27: A solution with extensible sets
Stability of aspects can be improved by following various design rules: the pointcuts should refer only to names that are available in the interfaces of the modules that are imported by the aspect, they should use only name patterns that rely on very stable naming conventions.

Another problem is that aspects can change the behaviour of a module in arbitrary ways, what can violate the internal assumptions of the module and lead to unpredictable results. A similar problem exists also in case of inheritance, because for each class we can define subclasses that can override the base methods in arbitrary ways. Nevertheless, object-oriented languages provide various means to limit the possibilities of extension. For example, in Java methods declared as final cannot be overridden, while the private members of a class are even not visible to the subclasses. Similar mechanisms could be provided to constrain the possibility of overriding with aspects.

The problem of controlling the access of aspects to module joinpoints are addressed by the Open Modules of Aldrich [Ald05]. It is proposed that the calls to the public functions of a module from outside should be accessible to aspects, while calls inside the module are hidden from the aspects, unless they are exported explicitly by providing a corresponding pointcut in the interface of the module. However, in object-oriented languages the notion of external calls is not that clear: for example, it is not clear how to treat calls between objects of the same class and calls between classes related by inheritance relations. This is even more complicated in case of virtual classes, because we have classes inside classes that are related by explicit and implicit inheritance relations.

Open Modules is just a formalism that provides only call pointcuts with no means for quantification: the pointcuts are defined by explicitly referring to individual functions or other modules. It does not tell how to combine quantification with explicitly exported pointcuts. This problem is addressed in Open Modules for AspectJ [OAT+06] that are based on a somewhat different approach: the pointcuts exposed by modules (sets of classes) serve as additional filters to the joinpoints selected by the pointcuts of aspects. This solution allows the aspects to quantify in an arbitrary way and still not violate the constraints of the modules. On the other side this can also be a problem, because in order to understand the actual effect of an aspect its pointcuts must be analyzed in combination with the pointcuts of all other modules that it can potentially advise.

Another problem addressed by the Open Modules for AspectJ is that it is necessary to provide different access to modules for different aspects. For example, aspects for profiling and debugging need to access much more joinpoints than the aspects that implement some specific functional feature. It is proposed that some of the aspects can be declared as friends of the module and this way be allowed to access all joinpoints of the module. Further, modules can be hierarchically grouped into larger modules that can modify the visibility of their children and add new friend aspects. The problem is that the set of friends is closed and must be known to the developers of the modules.

Clifton and Leavens [CL02] argue that certain kind of aspects, called spectators, do not change the behaviour of the modules that they advise and, therefore, do not create problems to modular reasoning. So there is no need to
constrain the set of joinpoints that can be accessed by such aspects. While, the aspects that can change the behaviour of a module (a set of classes) must be explicitly declared as its assistants, which are analogous to the friends in the Open Modules approach. A spectator aspect should not change the behaviour of the code that it advises: neither directly by changing the control flow nor indirectly by changing the state that it does not own. However, the authors note that verification of this property is undecidable in a general case and propose a practical approximation instead: the advices of a spectator aspect should not throw unchecked exceptions, and its around advices must call proceed once and leave its arguments and return value unchanged.

Another approach is proposed in CEAOP [gpc06]: instead of classifying aspects, individual events are classified into skippable and non-skippable. Non-skippable events can only be observed by aspects, while the skippable events can also be skipped, which means that it is possible to change the control flow at these events. [NN07b] proposes a component model for concurrent aspects, in which abstract skippable and non-skippable events are declared in the interfaces of aspect that use them, while connectors bind the abstract events of aspects to the concrete events of the components. It is however not possible for components to declare which of their events are allowed to be skipped.

**Proposal**  We propose to combine the ideas of Clifton and Leavens with the idea of skippable events and apply these ideas to our proposals for better modularization of features. In particular, we propose that a class (also a family class) should explicitly expose the jointpoints (or events), at which the clients of the class can modify its behaviour - we will refer to them as open joinpoints. Further, we propose to distinguish between the extensions of the clients that modify the behaviour of the class and the extensions that only observe the behaviour. The modifying extensions can be applied only to the open joinpoints exposed by the class. On the other hand, the extensions that do not modify behaviour can be applied without constraints.

The classes can extend other classes by inheriting from them and overriding their methods or by advising the methods with the pointcut/advice mechanism. Both kinds of extensions may modify or not modify the original behaviour. For distinction between modifying and observing advices we can follow the approximation proposed by Clifton and Leavens: a non-modifying advice should not throw unchecked exceptions, and if it is an around advice it must call the proceed once without modifying its arguments and return value. The same rule can be applied for method overridings: a non-modifying method overriding should call its super method once and do not modify the arguments and the return value. We also propose to distinguish between modifying and non-modifying extensions at the granularity of individual methods and advices rather than at the level of entire aspects (classes).

Further we need to clarify the concept of clients and implementers of a class, because classes can also be abstract and implemented by their subclasses. In Sec. 2.3.1 we introduced such distinction for inheritance relations: the clients of
a class inherit from it through the requires relation, while the implemeters of the class inherit through the extends relation. Of course, a class can be used not only by inheritance, but also by aggregation, instantiation or type references. All the modules that use the class in that way would also be considered as clients of the class. A family class will see its virtual classes as an implementer, while a virtual class will see its family class and siblings as a client. Finally, a class is also considered as an implementer of itself.

By the joinpoints of a class we mean all joinpoints, the shadows of which are lexically within the definition of the class. The internal joinpoints of a class are the joinpoints of the class as well as the joinpoints of the classes that it sees as an implementer. The external joinpoints of a class are all other joinpoints, i.e. the joinpoints of the classes that it sees as a client.

Classes see their internal joinpoints as open, which means that they can extend them without restrictions. Open joinpoints can be exposed to clients in two different ways. First, some of the public methods of a class can be marked as open. Second, classes can expose their internal joinpoints to their clients in form of open pointcuts. The pointcuts marked as open are allowed to select only open joinpoints, i.e. the internal joinpoints of the class in which they are declared and the external joinpoints that are exposed as open. In interfaces (abstract classes) we can declare abstract open pointcuts that can be specified in the implemeters of the interface. Open pointcuts can also be extended and composed in the way described in Sec. 2.3.3.

It must be possible to check statically and in a modular way if a pointcut declared as open selects only the joinpoints that it sees as open. By a modular checking we mean that the checking algorithm should use only the class itself and the set of explicitly imported and inherited classes. For example, if a pointcut selects methods by a pattern \* \*.foo(..), it cannot be checked in a modular way if all selected methods are indeed open, because for this we must know about all the classes of the program. However, for methods selected by a pattern \* Foo.\*(..), where Foo is a class visible to the pointcut, we can check if the pointcut sees this class as an implementer or if all the methods of the class are declared as open.

Further, we also need to define rules for composition of open pointcuts. The disjunction of two pointcuts is an open pointcut only if both pointcuts are open. A conjunction of an open pointcut with any condition is again an open pointcut.

Our solution provides a fine-grained control over the extensions that can modify the behaviour of a module. Each module explicitly exposes the joinpoints at which the control flow can be changed, which prevents clients from changing the assumptions of the module in unexpected ways and in this way help to reason about the properties of the module in a modular way. On the other hand, we leave full flexibility to write aspects that do not need to change control flow, such as logging and profiling. This solution integrates well to virtual classes and the other extensions proposed in the document.

---

3Since only public methods can be open, the modifier open would imply that the method is public.
4We also must take to account private inheritance described in Sec. 2.3.2.
2.3.5 Stateful Aspects

The extensions proposed so far assumed an AspectJ-like pointcut language. However, as was explained in Sec. 2.2.3 such pointcut languages are not expressive enough to relate different events occurring during program execution. Therefore, we propose to extend our language with support for EAOP. As a result we would have EAOP abstractions in interfaces and make them extensible and composable.

In order to embed EAOP into CaesarJ and make it possible for an aspect to take into account the history of the computation, we propose to add a new kind of class member: states. States are defined using an FSP-like syntax, as in Baton (see Section 2.2.3.2). As states are defined individually, an EAOP aspect can be defined piece by piece.

For instance, the following class:

```java
class Server {
    abstract state Session(Customer c);
    state Server() = ( login(Customer c) -> Session(c) );
    ...
    pointcut login(Customer c) :
        call(void Customer.login()) && this(c);
}
```

implements part of our previous Consistency aspect (see Figure 2.5). The state `Session` is not defined and is therefore declared abstract. It must be implemented in a subclass or a mixin.

2.3.5.1 Syntax

The syntactic definition of states is given by the following grammar (details of the concrete syntax may vary):

```
StateDef ::= state StateId(FormalArgs) = StateBody
         | abstract state StateId(FormalArgs);
StateBody ::= ( Branch ( | Branch )* )
Branch ::= ( AtomicAspect -> )+ StateId(ActualArgs)
AtomicAspect ::= Event => Advice
Event ::= PointcutName(Args)
Advice ::= ( Action; )* 
Action ::= JavaStatement
         | proceed(ActualArgs)
         | if JavaTest then Advice else Action
```

A state definition may be concrete or abstract, but declares the state arguments. The body of the state is defined via the choice `|` and the sequence `->` operators. The transitions are labelled by atomic aspects comprising an event
and a possibly empty advice. Each argument of an event can be either formal or actual. In the first case, an occurrence of the event initializes the variable. In the second case, matching takes place and decides whether the transition should take place or not. The advice language is restricted so that it is known, for each path, whether the advised joinpoint will proceed or not.

### 2.3.5.2 State instantiation

The initial state is chosen by an explicit instantiation in the class constructors. We assume that a single stateful aspect is instantiated for each object.

### 2.3.5.3 Inheritance

As other members, states may be redefined via inheritance, possibly using the keyword `super` to prefix or add choices to the definition available in the superclass.

This may look very permissive, as the behaviour of a stateful aspect may drastically vary from a superclass to a subclass without any constraint on behavioural consistency. Note however that, firstly, there exist several notions of behavioural consistency when considering objects (see, for instance, [LP05]) and, secondly, our language does not constrain the behaviour of standard objects, so why should their aspectual behaviour be constrained? This could be reconsidered if we wanted to define classes with behavioural interfaces.

It is also possible to define some strategies through programming conventions (which could be enforced by specific tools). For instance, one can imagine to only extend states by adding new branches, a state $S$ in a superclass is then defined by a choice between the previous definition of $S$, `super.$S` and one or several new branches. This is akin to linear substitutability, which states that every trace in a superclass is a trace in its subclass.

### 2.3.6 Dynamic Composition of Classes

**Motivation** CAESARJ supports only static mixin composition, which means that the compositions of classes modularizing features must be defined statically. However, which of the predefined compositions of features must be instantiated, can be decided at runtime. In this way we can have multiple predefined variations of components, which can be selected at depending on runtime conditions.

The problem is that the number of possible compositions grows exponentially with respect to the number of independent variation points. Therefore, the solution, which requires to define statically all possible compositions of features, does not scale.

Besides, the requirement to predefine all possible compositions of independent variants is also a limitation to modularity. When we have two independent
variation points of a component, it should be possible to bind them independently. For example, consider a graph display component that can have two independent variation points: one describes variations of the display style of edges and nodes, while another describes variations of graph layout rules. It should be possible to select the style of nodes and edges without knowing about particular layout rules, as well as the other way around. Besides, it should be possible to extend an independent variation points with new variants: we should be able to add new styles of edges and nodes independently from the variants of graph layout.

Therefore, it would be useful to be able to compose optional and alternative features at runtime, which means that we need a form of dynamic mixin composition of classes. In the current implementation of CAESARJ not all compositions of classes are valid, because of possibility of incompatible method signatures. Besides, since CAESARJ supports abstract classes, composition of classes can also be abstract. In order to make dynamic composition type safe we must be able to check at compile time if a particular dynamic composition would never produce conflicts. Besides, we must also check if a composition of two classes is not abstract and thus can be safely instantiated.

**Proposed Solution** In order to be able to manipulate classes dynamically we must make them as first class values, which means that we must allow to assign classes to variables, pass them as parameters to methods and return them as result. Further, we must provide an operation to compose such class values. Dynamic mixin combination with virtual classes is already available in gbeta [Ern02], but gbeta is very different from Java, and providing dynamic composition for a Java based language poses new challenges to the language design as well as to implementation. In particular, abstract methods and classes, which are not available in gbeta, pose new challenges.

A low-end solution, which could be provided in the first version of the implementation platform, could rely on Java reflection capabilities. Java already allows treating classes as values of a special type `java.lang.Class`. Such classes can be instantiated over Java reflection API and their instances have `Object` as a static type. For composition of classes we could make a new API function, which takes two classes and produce a new class which is the mixin composition of the input classes. The resulting class then can be instantiated using reflection mechanisms. If the class is abstract then its instantiation would generate a runtime error.

A high-end solution would be to provide an appropriate type system which can determine the types of the instances of the composed classes and can statically check if a dynamic composition of two classes produces a concrete class. In the following we describe the elements of a language supporting dynamic composition of classes.

**Class Types** In order to be able to say more about class values, we must more precisely describe their type. Since our primary goal is to compose class values
and at some point instantiate the composition, the types of classes should tell us when a class value can be instantiated and what the type of its instances is. As was already mentioned only complete classes can be instantiated. In order to know if a composition of two classes is complete, we must know if they eliminate the requirements of each other. Therefore, it is essential that the class types completely describe what their values require and what they provide.

So the general form of class types is \([R_1 \& R_2 \& \ldots \& R_n \Rightarrow P_1 \& P_2 \& \ldots \& P_m]\), where \(R_i\) are the required classes and \(P_i\) are the provided classes. If the type does not have requirements, the arrow can be skipped. So the type has the form \([P_1 \& \ldots \& P_m]\). The values of such type are complete classes, i.e. they can be instantiated.

The most precise type of a concrete class \(C\) with (direct and indirect) requirements \(R_1 \ldots R_n\) is \([R_1 \& \ldots \& R_n \Rightarrow C]\), because the class provides itself but not its requirements, while the most precise type of an abstract class is \([R_1 \& \ldots \& R_n \& C \Rightarrow]\), because the class does not give any guarantees neither about its implementation, nor about the implementation of its superclasses. For the sake of conciseness, we could also provide a special syntax to include a class with its all requirements to a type. For example, \([C]\) could translate to \([R_1 \& \ldots \& R_n \Rightarrow C]\) if class \(C\) is concrete and to \([R_1 \& \ldots \& R_n \& C \Rightarrow]\) if the class is abstract.

The subtype relation of class types is a reflective and transitive closure over the following rules:

- If a class \(C\) extends classes \(P_1 \ldots P_n\) then \([C]\) is a subtype of every \([P_i]\).

- \([R_1 \& \ldots \& R_n \Rightarrow P_1 \& \ldots \& P_m]\) is a subtype of \([S_1 \& \ldots \& S_k \Rightarrow Q_1 \& \ldots \& Q_l]\) if for every \(Q_i\) exists \(P_j\) such that \([P_j]\) is subtype of \([Q_i]\) and for every \(R_i\) exists \(S_j\) such that \([S_j]\) is a subtype of \([R_i]\). This rule basically tells that a subtype can weaken the requirements and strengthen the provided part.

Note that the use clause does not have any influence nor on the type of the class, nor at its subtype relationships.

For the class values we will provide a composition operation: for class values \(a\) and \(b\), expression \((a \& b)\) produces their mixin composition, which is again a class value.

The type of composition of two values of types \([R_1 \& \ldots \& R_n \Rightarrow P_1 \& \ldots \& P_m]\) and \([S_1 \& \ldots \& S_k \Rightarrow Q_1 \& \ldots \& Q_l]\) is a type, which provides both \(P_1 \ldots P_m\) and \(Q_1 \ldots Q_l\) and requires all \(R_1 \ldots R_n\) that are not eliminated by \(Q_1 \ldots Q_l\) as well as all \(S_1 \ldots S_k\) that are not eliminated by \(P_1 \ldots P_m\). Here \(C\) eliminates \(D\) if \([C]\) is subtype of \([D]\).

**Restrictions on Virtual Classes** In order to support dynamic composition of classes, we must constrain extensibility of virtual classes, because in the current implementation of CAESARJ a composition of two concrete classes can produce

---

5For simplicity, we will assume that all complete classes can be instantiated with an empty constructor
an abstract class. An example of such situation is given in Figure 2.28: class MShape declares an abstract virtual class Shape, class MCircle declares class Circle, which is a concrete subclass of Shape, and class MDraw introduces an abstract method draw to Shape. All three classes, MShape, MCircle and MDraw are complete and can be instantiated, but the composition of MCircle and MDraw is not complete, because Circle, which a concrete virtual class, would have an unimplemented method draw.

Such incompleteness can be easily detected in case of static composition of classes, because at the composition we know all virtual classes and all their methods, so we can check if there are no unimplemented methods in concrete virtual classes. However, such checks are not possible in case of dynamic composition, because we compose classes without complete knowledge about their internals. For example, by our typing rules the composition below would be legal, but it would produce an abstract class:

```
[MShape] circle = MCircle;
[MShape] draw = MDraw;
MShape sh = new (circle & draw)();
```

The type safety of our rules relies on the assumption that the composition of any two classes does not create new requirements other than those that are already declared for the classes. Such assumption is not true if it is allowed to have independent extensions of abstract virtual classes with new abstract methods as well as with new subclasses. The problem is analogous to the problem of modular type checking of modules that support independent extensions with new data types as well as with new operations [MBC04]. In fact our problem can be solved by introducing analogous limitations as those proposed in the paper:

- Inside a concrete class all methods of its all abstract virtual classes must be implemented, except for those that were introduced in the original declaration of the virtual class and those that were inherited over the requires relationship. In this case there is no limitation on extensions with new subclasses of the virtual class.

- Alternatively, an abstract virtual class can be marked with keyword `sealed`, which means that all direct subclasses of the class must be declared in the module that contains the original declaration of the virtual class. In
this case there is no limitation on extensions with new abstract classes of the virtual class.

Note that these restrictions are very specific and they do not limit extensibility much. For example, there is no limitation of extending virtual classes with new concrete methods or extending concrete virtual classes with new subclasses.

In fact, the first rule forces to give default implementations for new abstracts method of an abstract virtual class in order to avoid that this method can remain undefined in some concrete subclass. However, in certain cases a default implementation may be not possible and thus it could do nothing more than just to throw some runtime error. In this case it would be still useful to have a check if all statically known subclasses of the virtual class provide an implementation of that method. Therefore, we could mark such method with a special keyword, which tells that compiler must check if this method is overridden at least for the statically known subclasses.

Other Issues In order to enable safe dynamic composition of classes, we must ensure that the compositions cannot lead to conflicts. This is not the case in the CAESARJ implementation of virtual classes and mixin composition, because it generates an error when classes with incompatible method signatures are composed. Such problem can be avoided by following the rules of hygienic mixins [FKF99, ABC03]: methods are composed only if they originally stem from the same superclass, in other cases they are simply considered as two unrelated methods. The method calls are resolved by the static type of the receiver expression.

As was already mentioned, only class values with no requirements, i.e. of some type \([P_1 \land \ldots \land P_m]\), can be instantiated. An instance of a class of such type is an instance of all the classes \(P_1\) and \(P_m\). Thus in order to describe the types of such objects precisely we must introduce so called intersection types [CDC80]. An intersection of types \(T\) and \(U\) is a greatest lower bound of these two types. In other words it is a type, which has all features of \(T\) and \(U\) and each subtype of both \(T\) and \(U\) is also a subtype of their intersection.

We can also consider an integration of class types to the Java reflection mechanism. For example, the class types could implement the interface of \(java\_lang\_Class\). Further, we could provide an expression analogous to \(e\_class\), which returns the class of the object to which \(e\) evaluates. The static type of such expression could be inferred from the static type of the expression \(e\).

2.3.7 Dependent Classes

Motivation In our survey of D3.1 [PRG+07] we have identified the lack for high quality large-scale parameterization mechanisms. In object-oriented languages parameterization at the scale of classes (objects) and programs can be achieved by correspondingly object fields and global variables. The variables can, however, be used only inside method bodies and, thus, can influence only
behaviour of classes, but not their structure and interface. Such variations can be expressed by conditional compilation in C++ as well as various frame and template languages, but problem with these mechanisms is that they do not have the typical qualities of programming languages, such as modular checking (of syntax and types) and incremental compilation. A high-quality parameterization mechanism available at the level of classes is generics, but the expressivity of generics is very limited, because the interface and implementation of generic classes cannot depend on the generic parameters.

A virtual class can also be seen as a class parameterized by its family object: we can vary the interface and implementation of the virtual class depending on the type of the family object. The family object is passed as a parameter during instantiation of the virtual class, thus virtual classes provide a dynamic parameterization of classes. The problem with virtual classes is that they are too tightly bound to their parameter types and other virtual classes that depend on these parameter types. The reason of this is that a virtual class must be declared within its family class, i.e. within the type of its parameter, which also means that all virtual classes that depend on a particular parameter type must be declared within one module. Further, such parameterization is limited in a way that it supports only one parameter. It is not possible to declare virtual classes that depend on multiple families.

**Proposed Solution**  As a solution to the problem of large-scale parameterization we propose the idea of dependent classes: the classes the implementation and interface of which depend on one or more explicitly declared parameters. Since classes can be used as modules, dependent classes can also be used for describing parametric modules. The idea and semantics of virtual classes was developed in the context of the project and the results of this work is published in [GMO07].

Dependent classes generalize virtual classes by allowing the classes to depend on multiple parameters. Further, the parameters are declared explicitly rather than by nesting, which solves the problem of coupling between the virtual classes that depend on the same parameter types. The parameters of dependent classes are bound during their instantiation, which means it is a dynamic variation mechanism. In the implementation of the class the parameters are available as immutable object fields.

The dependency on parameters is described by a dispatch mechanism rather than by conditional statements within the definition of the class. A dependent class can have multiple declarations with different types of the parameters. Each declaration specifies the specific implementation and interface of the class that is available for these parameter types. During instantiation of dependent class all its declarations that match the dynamic types of the constructor parameters are collected and composed - the composition then determines the interface and the implementation of the instantiated object. The major advantage of the dispatch mechanism is possibility to extend the class with new variants: new declarations of the dependent class can be added that define its interface and
implementation for new specific cases.

In the description of type of an object a dependent class, we can specify not only the name of the class of the object, but also the type of its parameters. Since the parameters represent variations available for the class, the type of the object can precisely describe its configuration. The types of the parameters can be classes or paths. A path is an expression limited to navigation over object fields (including parameters). The type described by such expression is a singleton type, which contains only the value of the expression. In this way we can express a form of dependent typing, which allows to express covariant relations between objects, e.g. that the objects belong to the same family or have the same property.

2.3.8 Domain Specific Languages

The extensions proposed so far should significantly improve the capabilities of CaesarJ with respect to supporting variability in software product lines. However, such a language remains a general-purpose programming language. Some trade-offs have to be made and some features left aside. We think that a good way to further extend such a kernel language is to layer domain specific languages (DSLs) on top of it. In order to focus the work, we plan to consider two main directions: support for concurrency and distribution, and support for dynamic configuration and creative construction.

2.3.8.1 Support for concurrency and distribution

Nowadays, most applications are distributed (and therefore concurrent). Our case studies are no exception to this fact. At some level in the generation of an application, this has to be taken into account and proper language support has to be provided so that both humans and programs can properly reason about this difficult concern. Support can be provided at different levels. We are here interested in the support that we can provide at the programming language level. Our objective is to reduce the semantic gap which may exist between some high-level architecture of a product line and its implementation, in order to reduce the overall complexity of the implementation. In this context, if we consider, for instance, that mapping a crosscutting concern onto an aspect is a good idea, then we also need distributed aspects (as well as distributed components for non-crosscutting concerns).

Our previous work on concurrent and distributed aspects as well as on invasive patterns (see Section 2.2.3.2 and 2.2.3.3) has covered a number of interesting language constructs that make it possible to better modularize concurrent and distributed software, and provide architectural support that directly maps onto implementation. Rather than hardwiring these constructs into our “kernel” implementation language, we suggest to adopt a layered and modular approach and provide ways to extend this kernel language with concurrent and distributed facilities by designing and implementing composable DSLs. Each
DSL implements a specific set of constructs (local synchronization, remote synchronization, groups, patterns...). These DSLs are then composed together and with the kernel language to provide the appropriate concurrent or distributed language. The assumption here is that program transformation is used to compile the DSLs into kernel language code. Intermediate layers may also appear (in the same way as invasive patterns are compiled into AWED, which is compiled into Java).

Such an approach has a number of advantages over a monolithic approach:

- The implementation language is open and extensible.
- The implementation language can be configured with respect to the product family (the implementation is itself a product family).
- The design and implementation of the kernel language and of the extensions are decoupled.
- Although we plan to use grammarware to implement the DSLs (grammarware currently provides better support for concrete syntax and semantic analysis), we may decide at a later stage to turn to modelware and move the level at which models are turned into code.\(^6\)

It may also have drawbacks as it may actually lead to opening a semantic gap between the kernel language and the full language with, for instance, cryptic error messages and efficiency issues. Closing this gap may require then to reconsider the design and implementation of both the kernel and the extensions. But is is likely that the modifications would be localized without the necessity to backtrack to a monolithic approach.

### 2.3.8.2 Dynamic configuration and creative construction

Reducing the gap between configuration entities (features, concerns, components...) and runtime entities, apart from reducing the complexity of the implementation, also makes it easier to consider dynamic configuration. Static switches are turned into dynamic switches and configuration conditions may depend on runtime values. One step further consists in considering reconfigurations, which makes it possible to deal with runtime evolution and adaptive systems. It is then natural to consider a DSL dedicated to dynamic configuration (and possibly reconfiguration). If both static and dynamic configuration is required, two DSLs must be provided. One for static configuration and one for dynamic configuration. These DSLs are very similar but embedded in very different contexts. Still, it seems that they could be implemented in a concerted way as the composition of a core configuration DSL parameterized by a DSL describing the configurations conditions and proper semantic analysis switches (in order to produce code with the same global structure but differing in a number of details depending on whether static or dynamic configuration is considered).

\(^6\)This could mean, at least partially, self-applying AMPLE technology.
Reconfiguration requires an additional DSL for accessing the architectural elements of interests (see for instance FPath [DL06], which can be seen as a structural pointcut language) as well as another DSL providing the primitives (delete, add...) necessary to alter an architecture (this is an advice language).

Such a combination of static and dynamic configuration opens the way to interesting scenarios with respect to the use of DSLs for creative construction [CE00] of applications where simple name-value based configuration does not suffice.

In D3.1, SAP has given examples in for such DSL in the context of enterprise software. For instance, Business Process Execution Language (BPEL) [OAS07] and Business Process Modeling Notation (BPMN) [Obj06] allow customers to configure business rule engines of SAP systems according to their business processes at runtime because every business is essentially different. Most companies draw their competitive advantages out of subtle deviations from standard business processes. These variations often go beyond simply enabling/disabling switches or changing parameter values. Business experts need means for “programming in the large”, i.e., wiring state transitions and message-based process interactions, and “programming in the small”, i.e., being able to model conditional and/or parallel execution of business process steps, ideally supported by graphical tools.

To address the challenge of supporting flexible binding times of features, it should be possible to interpret such a DSL at runtime in cases where full flexibility is required. At the same time, it should be possible to bind this variability at build time if the corresponding flexibility is not required in a specific customer scenario. This optimisation would save valuable execution time, thus increasing system responsiveness.

Note however that, depending on the level of the configuration, different choices of technology (grammarware, modelware, or a combination of both) may be required.

2.3.9 IDE Support

In the previous sections we describe a set of proposals for improving existing programming languages. The outcome of implementation of these proposals will be a new programming language, implemented by a compiler and an appropriate execution environment. However, implementation of a programming language in a broader sense means also providing a set of appropriate development tools, usually in a form of an integrated development environment (IDE). In the AMPLE project, we have selected Eclipse [ecl] as a basis IDE, because of its explicit support for extensibility and the fact that most of existing tools of the project partners are based on Eclipse.

The first step of integration of a new programming language to Eclipse is integration of its compiler and execution platform. It should be possible to create projects and artefacts of that programming language, which are then automatically compiled by its compiler. It should be possible to manage compilation
dependencies and provide all resources that are necessary for compilation. Further the IDE should provide facilities to start the compiled program in the appropriate execution environment.

Integration of an incremental compiler is more complicated, because the IDE must track changes in the artefacts of the programming language, analyse the compilation dependencies between these artefacts and determine which of the artefacts must be recompiled.

A lot of development time is usually spent for debugging, therefore a debugging support is also an important aspect of implementation of a programming language. Since our programming language will be translated to the Java bytecode, it will be possible to use the existing Eclipse debugger for Java. However, in order to enable debugging the compiler must generate an appropriate debugging information, which includes the relation of bytecode to the corresponding lines in the source code as well as information about variables that can be inspected at various points in the program.

For a better debugging support it is also necessary to adjust the functionality for setting breakpoints, stepwise execution as well as inspection of the program state: the call stack and the heap. For breakpoint setting it is necessary to parse the source code and identify the points, at which the breakpoints can be set. The available stepwise execution functionality in Eclipse should be adjusted in order to jump over the fragments of code that are generated by the compiler. The inspection of program state also may need various adjustments, because the Java debugger would show the elements of the generated bytecode (methods, classes, variables, etc.), which wouldn’t completely correspond to the structure of the source code: certain source code elements may be represented by multiple elements in the bytecode, the names of the elements may be changed and a part of the generated code would be glue code that should be completely hidden from the developer.

IDE can help to visualize the structure of the implementation artefacts and allow navigating over it. The structure of an individual artefact can be visualized by highlighting syntax in the editors and providing an outline view. The package view of the Eclipse provides a hierarchical view over the entire program. Further views can be provided to display other relationships between the structural parts of the program: for example the relationships between classes and links between pieces of advice and their joinpoint shadows. Further, the views can be made interactive in order to enable navigation over different relationships between the structural elements over program. A further help for navigation would be a tool that allows searching for different program elements by various logical criteria.

The IDE tools are also important for traceability support, because the views of the implementation artefacts can be extended with links to the artefacts that define the requirements and architecture of the program. In the context of MDD some of the implementation artefacts are generated from higher-level model. IDE may visualize the relationships between the source and the target artefacts and enable navigation between them. Further, IDE may automatically update generated artefacts after the changes in the source models, and protect parts
of the generated artefacts from manual changes. The traceability links can also be used for relating the runtime entities exposed by debugger to the entities in the models from which they are generated.

### 2.3.10 Expressing Variability in Structural Models

This section describes concepts and tools that support the definition of feature-based variability in structural models and hence the selective adaptation of models. Structural models are models built with a creative construction DSL. Features expressed in structural models can be linked to configuration models. This enables the adaptation of those structural models based on the feature selection in configuration models.

#### 2.3.10.1 Positive Variability

As illustrated in Figure 2.29, positive variability [Cop98] starts with a minimal core and selectively adds additional parts. The core represents the model parts that are common to all products within the product line. Varying model parts are attached to the core based on the presence or absence of features in the configuration models.

When expressing variability in models, optional model elements have to be connected to the core at specific points. This is analogous to the concept of join points in aspect-orientation. In modeling, join points are elements of the modeling language which aspects coordinate with. Similar to weaving of code level aspects in traditional AO languages, aspects are defined on model level and are composed with a base model. Weaving is technically done by an aspect weaver at designated join points.

In the field of AOP, there are various join point models around and many are still under development. Join point models heavily depend on the underlying programming language and AO language [AOS07]. Hence, the join point model of AOM depends on the underlying modeling language and the language used to express aspects on model level. We introduce a model weaving approach including a join point model that supports the expression of aspects on model level and a composition technique for base models and aspect models.

Figure 2.30 illustrates the concept of model weaving. A given base model ($M_A$) and an aspect model ($M_{\text{Aspect}}$) are composed. The aspect model consists of pointcut definitions that capture the points where in the base model additional model elements should be added. In addition to the pointcut definitions, the aspect model contains advices that represent the elements to be added. The
Figure 2.30: Model weaving

aspect model uses both name matching and pointcut expressions to select the desired join points.

In the context of the project we have developed a tool called XWeave\(^7\) [GV07a] [GV07b] that implements the concepts presented above. It is based on Eclipse as a tool platform, Ecore as the meta meta model. Since it is an oAW workflow component, it easily integrates into oAW based model processing.

XWeave is a model weaver that can weave models that are either instances of Ecore (called meta models) or instances of these meta models (called models). The tool takes a base model as well as one or more aspect models as input and weaves the content of the aspect model into the base model. Weaving an aspect element means that all properties of the element including its child elements are woven into the base model. Both aspect model and base model must conform to the same meta model. The aspect model consists of definitions that capture the points where in the base model the additional model elements should be added (the pointcut in AO terms). It also contains these additional model elements (the advice in AO terms). This is a form of asymmetric AO. During weaving, aspect elements are physically woven into the base model. The result of this process is an updated model. Subsequent tooling cannot tell the difference between a woven and a non-woven model.

The XWeave approach can be applied to both problem space models and solution space models. This means that both domain models and software models can be configured.

**Linking XWeave to Configuration Models** Model weaving assists in the composition of different separated models into a consistent whole. It allows to capture feature-dependent parts of models in aspect models. This technique supports a clear separation of variable model parts from the core and supports an automatic composition to create a complete model representing a member of the product line.

The dependency between aspect models and features is specified in the oAW workflow. Aspects that implement optional parts of structural models are linked to features defined in configuration models. Based on a selection of features, the corresponding aspect models are woven to the base model. This dependency between features and aspect models is illustrated in Figure 2.31.

\(^7\)XWeave is part of openArchitectureWare 4.2 and can be downloaded from http://www.openarchitectureware.org/
2.3.10.2 Negative Variability

As illustrated in Figure 2.32, negative variability selectively takes away parts of a creative construction model based on the presence or absence of features in the configuration models. This technique is fundamentally different to the technique introduced in the previous section. When using negative variability to implement feature-based variability in structural models, one has to build the "overall" model manually and connect elements to certain features in a configuration model. The model is then tailored based on a certain feature selection, thus model elements are taken away from the full model.

A tool called XVar\(^8\) [GV07a] has been developed that implements the concept of negative variability. Like XWeave, it is based on Eclipse as a tool platform, Ecore as the meta meta model, and oAW as the tool for model processing.

Figure 2.33 illustrates how XVar links structural models to configuration models. A dependency model captures the relationships between model elements and features. Depending on the selection of features, the structural model is tailored to only contain the model elements needed for the respective configuration by deleting model elements whose features are not selected. Deleting means that the element itself and all references to this element are removed from the model.

XVar can tailor models that are either instances of Ecore (called meta models) or instances of these models (called models). The dependency between features and model elements is specified in a separate dependency model. The advantage of defining feature dependencies in an external model is that no in-
vasive changes to the model are required. All dependencies are explicitly listed in the dependency model. The XVar approach can be applied to both problem space models and solution space models.

2.3.11 Expressing Variability in Code Generation Templates

Motivation A generator generates some textual output (e.g. code, build scripts, XML configuration files) from a model. The generator operates on the meta model of the DSL and thus has to know this meta model. Figure 2.34 illustrates the relationship between model, meta model, and generator.

A template language including a suitable template engine is used to generate the output. Templates support the generation of any text-based output.

In our approach code is generated from the final solution space model. As meta model, models and the respective transformations can vary, we also need to incorporate variability into the code generation process.

Proposed Solution For dealing with these kind of variations we propose to use AO on code generation level. In the context of the project we extended the template language Xpand [ope07b] with support for template aspects.

XPand supports around advices whereby any execution of a template definition can be a join point. The definition name part of a pointcut must match the fully qualified name of the join point's definition. The asterisk character is used to specify wildcards.

Also, we want generator advices only to be applied in case a certain feature is selected in the current configuration. The dependency between template aspects and features is specified in the workflow (cf. Figure 2.35).

Applying aspects to code generation templates realizes positive variability on generation level. Again, as an alternative, one could also develop the overall generator and exclude templates in case a certain feature is selected.

2.3.12 Expressing Variability in Model Transformations

Motivation In model transformations a model is transformed into another model, the input model is typically left unchanged. In Figure 2.36 a model \( M \) is transformed into a model \( K \). Both models are instances of different meta models. An important advantage of model transformations is that a clean separation
There is also the notion of a model modification, where a model is modified 'in place', i.e. the input model is changed, and no additional output model is created. Since such a model modification is technically almost identical to a model transformation as defined above, this section focuses exclusively on model transformations.

As demonstrated in the previous section, both models and meta models can vary using the concepts and tools we developed. This directly leads us to the need of varying model transformations. New meta model elements brought by additional features must be added to the transformation workflow. Again, the adaptation of model transformations is only required in case the respective features are selected in the current configuration.

**Proposed Solution** We propose to solve this problem by applying AO to model transformations. In the context of the project we have extended the transformation language Xtend [ope07b] with support for aspects. Xtend supports the application of advices to model transformation functions. Only around advices are supported.

As we want to apply advices only in case a certain feature is selected in the current configuration, we need to link advices to features. Figure 2.37 illustrates how this is done. Transformation aspects are connected to features in the oAW workflow. The advice is then only applied to the transformation in case the respective feature is selected.

Applying aspects to model transformations realizes positive variability on trans-
Figure 2.37: Variability in model transformations

formation level. As an alternative, one could also develop the overall transformation and exclude transformation steps in case a certain feature is selected.

### 2.4 Integration of the Implementation Platform into the AMPLE Process

This section explores and explains the relationships between the implementation platform to be produced in WP3, the traceability framework being defined in WP4 (traceability), and the architectural design approach currently defined in WP2 (architecture), in order to ensure that the implementation platform integrates properly into the general picture of the AMPLE project.

#### 2.4.1 Integration with the Traceability Framework

The traceability framework, currently being defined in WP4, does not make any assumptions about the artefacts being traced: they can be model elements defined as instances of a metamodel, informal requirement documents, textual configuration scripts or textual code written in a traditional programming language. The only requirement considered by the traceability framework is that there should be a mechanism for referencing and finding traceable artefacts in an unique way, i.e. some kind of unique identifiers should be provided for traceable artefacts.

However, to define what are the traceable artefacts in the implementation platform is beyond the scope of WP3. Identification of traceable artefacts must be driven by the traceability requirements and scenarios already identified in WP4, and the way how this tracing information is generated is responsibility of the task that covers the transformations from architecture to implementation. As soon as the traceable artefacts are created, the corresponding information must be loaded in the traceability repository.

Therefore, the only responsibility of WP3 is to define how these traceable artefacts can be referenced and localized in an unambiguous way, and what in-
formation need to stored in the traceability repository according to the identified traceability requirements.

It should be noted that in order to achieve this task, an explicit metamodel for the implementation platform/language is not absolutely necessary.\(^9\) It is possible to trace to a textual language without an explicitly defined metamodel (see [ON06, OO07] for some examples).

It is worth to mention that the tool support for the implementation platform would need to deal with code automatically generated from models, which must not be modified manually. Therefore, these tools should ensure that automatically generated code is not changed manually.

### 2.4.2 Integration with the Architectural Design

An approach that produces architectural design models for Software Product Lines has already been defined in WP2 [SGF+07]. These architectural design models serve as input for automatic model transformations that generate implementation models (code) conforming to the languages of the implementation platform defined in WP3.

Although one of the most interesting contributions of model-driven techniques is that the language or metamodel used at one development phase (e.g. architecture) must not enforce the use of any particular language or metamodel for either the preceding (e.g requirements) or the succeeding development phases (e.g. implementation), and the gap between the different languages or metamodels is bridged by means of automatic model-transformations. This allows developers to focus on the goals relevant to a certain development stage, without worrying how the created artefacts at one development phase are mapped to the artefacts of the next development phase. However, a big gap between the language used for architectural design and the language for the implementation platform could imply a considerable effort in the development of the automatic model transformations from architecture to implementation that could not be affordable inside the time frame of the AMPLE project with the currently assigned resources. Therefore, we need to ensure the generation of the automatic model transformations from architecture to implementation is a realistic task inside the AMPLE project context. We briefly analyze how this task can be performed.

Architectural models produced using the design approach of WP2 are UML 2.0 models that make a strong use of the UML merge composition mechanism between UML packages as a variation mechanism. The target language will be an improved version of CAESARJ, an aspect-oriented extension to Java with a set of new constructs for dealing with feature-based decomposition. Initially, there should be no serious problems for generating Java code from the structural UML diagrams that comprise the architectural description. In addition, UML packages encapsulating variants seems initially to nicely map to CaesarJ family

\(^9\)The modelling languages defined for requirements and architecture have an explicit metamodel and they also need to define the traceable artefacts as well as how to reference and find them.
classes, promoting a seamless integration between both approaches. Therefore, there does not seem to be serious risk regarding the development of the model transformations from architectural models to implementation models in the context of the AMPLE project. These transformations would be implemented using the XPand language\(^\text{10}\), because several partners of the project have previous experience with it.

These transformations are also responsible of writing the corresponding the traceability information into the traceability repository according to the identified traceability requirements. It implies the creating and filling information about the traceable artefacts created during the transformation process as well as creating the information about the trace links between architectural models and implementation models, according to the identified traceability requirements. This can performed by implementing a tracing aspect in XPand. This aspect would create the corresponding trace model in the traceability repository in parallel to the generation of the traceable artefacts.

### 2.5 Conclusions and Outlook

In this chapter we proposed a set of various extensions and combinations of the technologies of the partners that together would provide a better implementation platform for the artefacts of SPL. As the central part of the platform we see an aspect-oriented programming language which integrates the features of CAESARJ and EAOP as well as the proposed improvements. The IDE support for the language would be available in form of Eclipse plugin that would integrate the compiler of the language and provide support for debugging, editing and navigation as described in Sec. 2.3.9.

The programming language would provide a lot of flexibility for decomposing software into modules. Virtual classes and propagating mixin composition enable decomposition of classes and their methods. Pointcut and advice mechanism enables crosscutting decomposition of the behaviour of the software. The proposed pointcut language would integrate the advantages of AspectJ and EAOP pointcuts (Sec. 2.3.5): AspectJ features would provide a lot of flexibility to select individual events, while EAOP would allow relating multiple temporally related events. Further, the language would also provide a lot of flexibility for decomposing pointcuts (Sec. 2.3.3 and Sec. 2.3.5).

The flexibility of decomposing software into modules is very important for implementation of SPLs, because it determines the flexibility of separating the code that is common to multiple products of a SPL from the product specific code as well as modularizing the parts of implementation that depend on independent variants or variation points. Such separation is a prerequisite for independent evolution of the product line and individual products. The flexibility of decomposition of software is also important in the context of MDD, as it may help to separate generated code from the manually written and to better

\(^\text{10}\)http://www.openarchitectureware.org/
align the structure of the generated code to the structure of the source models. Such alignment eases implementation of the transformations and avoids unnecessary bindings of variation at the modelling level. Besides, an alignment between the structure of source models and the structure of the generated code is necessary for enabling incremental generation and compilation as well as independent distribution and deployment of the generated code.

It must be pointed out that the quality of decomposition of software can be different. In D3.1 [PRG+07] we described multiple levels of separation and observed that most of the large-scale refinement mechanisms provide separation only at the structural level only, which means that dependencies and interfaces between modules are not explicitly declared. This causes a lot of problems. First of all, the software is difficult to understand because it is not clear what services a module provides and what are its relations to other modules. Further, it is not possible to type-check and compile the modules in an incremental way. Finally, this complicates evolution of modules: since there are no explicit dependencies and interfaces between modules each change can potentially destabilize all the system.

The programming language that we propose would provide a high quality of modularization without compromising the flexibility of decomposition. As the basis for our module system we take the family classes of CAESARJ, which already can express highly extensible modules with explicit dependencies and interfaces. The extensions that we propose would improve the quality of the modularization further, because they make it possible to describe different contracts for providers and clients of the features (Sec. 2.3.1), solve the problem of transitive dependencies of modules (Sec. 2.3.2) and provide an explicit control over the ways the clients of a module can change its behaviour (Sec. 2.3.4).

Besides the core programming language the implementation platform would also include various language extensions in form of DSLs. This is especially important in the context of product lines, because there can be language extensions that are useful in a particular domain, but not general enough to be included to a general purpose language. In particular, we are going to experiment with DSLs that are specifically designed for concurrency and distribution. Another design goal is to modularize the implementation of the general purpose language features so that they could be evolved independently and could be more easily integrated with other language features.

Most of variations in product lines can be bound at compile time, which is well supported by AOP and MDD, but there can also be variations that need to be bound at initialization time or runtime, which is only supported by a subset of AOP languages (cf. Sec. 1.8) and not by MDD. The proposal in Sec. 2.3.6 makes our mechanism for composition of family classes available at runtime. This enables to customize the features of a product according a configuration that is available during its startup. The composition is dynamic, but it can be checked statically that all possible combinations would produce valid programs. Therefore, such mechanism can be especially useful for supporting variability that is bound by the user during installation or configuration after the product is installed.
However, the composition at the level of classes does not support full runtime variability, because it does not allow changing the functionality of already instantiated objects. For example, it would be not possible to turn on and off individual features that extend the behaviour of other features. In such situations we can use dynamic aspect deployment and wrappers of CAESARJ, which were described in Sec. 2.2: dynamic aspect deployment enables dynamic extension of the behaviour of a program, while wrappers provide a way to extend the state of already instantiated objects. The dynamic aspects would also respect the extension interfaces described in Sec. 2.3.4. Besides, the modularization in family classes can be applied not only at the scope of entire program, but also at the scope separate components, i.e. for modularization of varying features of components. These components then can be instantiated at runtime and composed either by conventional object-oriented mechanisms or plugged-in in some event-based system or a dependency injection framework.

Another constituent part of the implementation platform is a set of tools and languages for implementation of model-to-model and model-to-code transformations. For this purpose we will use the openArchitectureWare framework, which provides a wide selection of tools supporting MDD. In the context of the project we have extended these tools with elements of AOP in order to support positive variability at the level of models (Sec. 2.3.10), model transformations (Sec. 2.3.12), and code generation templates (Sec. 2.3.11).
Bibliography


[DLBNS06] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In *Proceedings of GPCE’06* [gpc06].


[GBF07] Phil Greenwood, Thiago T. Bartolomei, Eduardo Figueiredo, Marcos Dósea, Alessandro F. Garcia, Nélio Cacho, Cláudio Sant’Anna, Sérgio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. On the impact of aspectual decompositions on design


